Flask Exceptional
中文翻译

Flask-PyMongo
中文翻译

Flask-Dashed
中文翻译

Flask/SQLAlchemy/Postgresgl
异步方案

Flask SQLAlchemy
中文翻译

flask-mail
中文翻译

Flask Testing
中文翻译

使用 Flask 设计
RESTful API

Flask WTF
中文翻译

Flask-Login
中文翻译

# 目錄

# Flask 扩展文档汇总

来源：Python中文学习大本营

# Flask-Babel

Flask-Babel 是一个 Flask 的扩展，在 babel, pytz 和 speaklater 的帮助下添加 i18n 和 l10n 支持到任何 Flask 应用。它内置了一个时间格式化的支持，同样内置了一个非常简单和友好的 `gettext` 翻译的接口。

## 安装

下面命令可以安装扩展:

```
$ easy_install Flask-Babel
```

或者如果你安装了 pip:

```
$ pip install Flask-Babel
```

请注意 Flask-Babel 需要 Jinja 2.5。如果你安装一个老的版本你将会需要升级或者禁止 Jinja 支持。

## 配置

在配置好应用后所有需要做的就是实例化一个 `Babel` 对象:

```
from flask import Flask
from flask.ext.babel import Babel

app = Flask(__name__)
app.config.from_pyfile('mysettings.cfg')
babel = Babel(app)
```

babel 对象本身以后支持用于配置 babel。Babel 有两个配置值，这两个配置值能够改变内部的默认值:

| | |
|---|---|
| `BABEL_DEFAULT_LOCALE` | 如果没有指定地域且选择器已经注册， 默认是缺省地域。默认是 `'en'` 。 |
| BABEL_DEFAULT_TIMEZONE | 用户默认使用的时区。默认是 `'UTC'` 。选用默认值的时候，你的应用内部必须使用该时区。 |

对于更复杂的应用你可能希望对于不同的用户有多个应用，这个时候是选择器函数派上用场的时候。babel 扩展第一次需要当前用户的地区的时候，它会调用 `localeselector()` 函数，第一次需要时区的时候，它会调用 `timezoneselector()` 函数。

如果这些方法的任何一个返回 `None` ，扩展将会自动回落到配置中的值。而且为了效率考虑函数只会调用一次并且返回值会被缓存。如果你需要在一个请求中切换语言的话，你可以 `refresh()` 缓存。

选择器函数的例子:

```
from flask import g, request

@babel.localeselector
def get_locale():
    # if a user is logged in, use the locale from the user settings
    user = getattr(g, 'user', None)
    if user is not None:
        return user.locale
    # otherwise try to guess the language from the user accept
    # header the browser transmits.  We support de/fr/en in this
    # example.  The best match wins.
    return request.accept_languages.best_match(['de', 'fr', 'en'])

@babel.timezoneselector
def get_timezone():
    user = getattr(g, 'user', None)
    if user is not None:
        return user.timezone
```

以上的例子假设当前的用户是存储在 `flask.g` 对象中。

# 格式化日期

你可以使用 `format_datetime()` ， `format_date()` ， `format_time()` 以及 `format_timedelta()` 函数来格式化日期。它们都接受一个 `datetime.datetime` （或者 `datetime.date` ， `datetime.time` 以及 `datetime.timedelta` ）对象作为第一个参数，其它参数是一个可选的格式化字符串。应用程序应该使用天然的 datetime 对象且内部使用 UTC 作为默认时区。格式化的时候会自动地转换成用户时区以防它不同于 UTC。

为了能够在命令行中使用日期格式化，你可以使用 `test_request_context()` 方法:

```
>>> app.test_request_context().push()
```

这里是一些例子:

```
>>> from flask.ext.babel import format_datetime
>>> from datetime import datetime
>>> format_datetime(datetime(1987, 3, 5, 17, 12))
u'Mar 5, 1987 5:12:00 PM'
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'full')
u'Thursday, March 5, 1987 5:12:00 PM World (GMT) Time'
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'short')
u'3/5/87 5:12 PM'
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'dd mm yyy')
u'05 12 1987'
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'dd mm yyyy')
u'05 12 1987'
```

接着用不同的语言再次格式化:

```
>>> app.config['BABEL_DEFAULT_LOCALE'] = 'de'
>>> from flask.ext.babel import refresh; refresh()
>>> format_datetime(datetime(1987, 3, 5, 17, 12), 'EEEE, d. MMMM yyyy H:mm')
u'Donnerstag, 5\. M\xe4rz 1987 17:12'
```

关于格式例子的更多信息请参阅 babel 文档。

# 使用翻译

日期格式化之外的另一个部分就是翻译。Flask 使用 `gettext` 和 Babel 配合一起实现翻译的功能。gettext 的作用就是你可以标记某些字符串作为翻译的内容并且一个工具会从应用中挑选这些，接着把它们放入一个单独的文件为你来翻译。在运行的时候原始的字符串（应该是英语）将会被你选择的语言替换掉。

有两个函数可以用来完成翻译：`gettext()` 和 `ngettext()`。第一个函数用于翻译含有 0 个或者 1 个字符串参数的字符串，第二个参数用于翻译含有多个字符串参数的字符串。这里有些示例:

```
from flask.ext.babel import gettext, ngettext

gettext(u'A simple string')
gettext(u'Value: %(value)s', value=42)
ngettext(u'%(num)s Apple', u'%(num)s Apples', number_of_apples)
```

另外如果你希望在你的应用中使用常量字符串并且在请求之外定义它们的话，你可以使用一个“懒惰”字符串。“懒惰”字符串直到它们实际被使用的时候才会计算。为了使用一个“懒惰”字符串，请使用 `lazy_gettext()` 函数:

```
from flask.ext.babel import lazy_gettext

class MyForm(formlibrary.FormBase):
    success_message = lazy_gettext(u'The form was successfully saved.')
```

Flask-Babel 如何找到翻译？首先你必须要生成翻译。这里是你如何做到这一点:

# 翻译应用

首先你需要用 `gettext()` 或者 `ngettext()` 在你的应用中标记你要翻译的所有字符串。在这之后，是时候创建一个 `.pot` 文件。一个 `.pot` 文件包含所有的字符串，并且它是一个 `.po` 文件的模板，`.po` 文件包含已经翻译的字符串。Babel 可以为你做所有的这一切。

首先你必须进入到你的应用所在的文件夹中并且创建一个映射文件夹。对于典型的 Flask 应用，这是你要的:

```
[python: **.py]
[jinja2: **/templates/**.html]
extensions=jinja2.ext.autoescape,jinja2.ext.with_
```

在你的应用中把它保存成 `babel.cfg` 或者其它类似的东东。接着是时候运行来自 Babel 中的 `pybabel` 命令来提取你的字符串:

```
$ pybabel extract -F babel.cfg -o messages.pot .
```

如果你使用了 `lazy_gettext()` 函数，你应该告诉 pybabel，这时候需要这样运行 `pybabel`:

```
$ pybabel extract -F babel.cfg -k lazy_gettext -o messages.pot .
```

这会使用 `babel.cfg` 文件中的映射并且在 `messages.pot` 里存储生成的模板。现在可以创建第一个翻译。例如使用这个命令可以翻译成德语:

```
$ pybabel init -i messages.pot -d translations -l de
```

`-d translations` 告诉 pybabel 存储翻译在这个文件夹中。这是 Flask-Babel 寻找翻译的地方。可以把它放在你的模板文件夹旁边。

现在如有必要编辑 `translations/de/LC_MESSAGES/messages.po` 文件。如果你感到困惑的话请参阅一些 gettext 教程。

为了能用需要编译翻译，`pybabel` 再次大显神通:

```
$ pybabel compile -d translations
```

如果字符串变化了怎么办？像上面一样创建一个新的 `messages.pot` 接着让 `pybabel` 整合这些变化:

```
$ pybabel update -i messages.pot -d translations
```

之后有些字符串可能会被标记成含糊不清。如果有含糊不清的字符串的时候，务必在编译之前手动地检查他们并且移除含糊不清的标志。

## 问题

在 Snow Leopard 上 pybabel 最有可能会以一个异常而失败。如果发生了，检查命令的输出是否是 UTF-8:

```
$ echo $LC_CTYPE
UTF-8
```

不幸地这是一个 OS X 问题。为了修复它，请把如下的行加入到你的 `~/.profile` 文件:

```
export LC_CTYPE=en_US.utf-8
```

接着重启你的终端。

# API

文档这一部分列出了 Flask-Babel 中每一个公开的类或者函数。

## 配置

`class flask.ext.babel.Babel(app=None, default_locale='en', default_timezone='UTC', date_fo`

Central controller class that can be used to configure how Flask-Babel behaves. Each application that wants to use Flask-Babel has to create, or run `init_app()` on, an instance of this class after the configuration was initialized.

`default_locale`

The default locale from the configuration as instance of a `babel.Locale` object.

`default_timezone`

The default timezone from the configuration as instance of a `pytz.timezone` object.

`init_app(app)`

Set up this instance for use with *app*, if no app was passed to the constructor.

`list_translations()`

Returns a list of all the locales translations exist for. The list returned will be filled with actual locale objects and not just strings.

New in version 0.6.

`localeselector(f)`

Registers a callback function for locale selection. The default behaves as if a function was registered that returns `None` all the time. If `None` is returned, the locale falls back to the one from the configuration.

This has to return the locale as string (eg: `'de_AT'` , “ `en_US` ”)

`timezoneselector(f)`

Registers a callback function for timezone selection. The default behaves as if a function was registered that returns `None` all the time. If `None` is returned, the timezone falls back to the one from the configuration.

This has to return the timezone as string (eg: `'Europe/Vienna'` )

## Context 函数

`flask.ext.babel.get_translations()`

Returns the correct gettext translations that should be used for this request. This will never fail and return a dummy translation object if used outside of the request or if a translation cannot be found.

`flask.ext.babel.get_locale()`

Returns the locale that should be used for this request as `babel.Locale` object. This returns `None` if used outside of a request.

`flask.ext.babel.get_timezone()`

Returns the timezone that should be used for this request as `pytz.timezone` object. This returns `None` if used outside of a request.

## Datetime 函数

`flask.ext.babel.to_user_timezone(datetime)`

Convert a datetime object to the user’s timezone. This automatically happens on all date formatting unless rebasing is disabled. If you need to convert a `datetime.datetime` object at any time to the user’s timezone (as returned by `get_timezone()` this function can be used).

`flask.ext.babel.to_utc(datetime)`

Convert a datetime object to UTC and drop tzinfo. This is the opposite operation to `to_user_timezone()` .

`flask.ext.babel.format_datetime(datetime=None, format=None, rebase=True)`

Return a date formatted according to the given pattern. If no `datetime` object is passed, the current time is assumed. By default rebasing happens which causes the object to be converted to the users's timezone (as returned by `to_user_timezone()`). This function formats both date and time.

The format parameter can either be `'short'`, `'medium'`, `'long'` or `'full'` (in which cause the language's default for that setting is used, or the default from the `Babel.date_formats` mapping is used) or a format string as documented by Babel.

This function is also available in the template context as filter named `datetimeformat`.

`flask.ext.babel.format_date(date=None, format=None, rebase=True)`

Return a date formatted according to the given pattern. If no `datetime` or `date` object is passed, the current time is assumed. By default rebasing happens which causes the object to be converted to the users's timezone (as returned by `to_user_timezone()`). This function only formats the date part of a `datetime` object.

The format parameter can either be `'short'`, `'medium'`, `'long'` or `'full'` (in which cause the language's default for that setting is used, or the default from the `Babel.date_formats` mapping is used) or a format string as documented by Babel.

This function is also available in the template context as filter named `dateformat`.

`flask.ext.babel.format_time(time=None, format=None, rebase=True)`

Return a time formatted according to the given pattern. If no `datetime` object is passed, the current time is assumed. By default rebasing happens which causes the object to be converted to the users's timezone (as returned by `to_user_timezone()`). This function formats both date and time.

The format parameter can either be `'short'`, `'medium'`, `'long'` or `'full'` (in which cause the language's default for that setting is used, or the default from the `Babel.date_formats` mapping is used) or a format string as documented by Babel.

This function is also available in the template context as filter named `timeformat`.

`flask.ext.babel.format_timedelta(datetime_or_timedelta, granularity='second')`

Format the elapsed time from the given date to now or the given timedelta. This currently requires an unreleased development version of Babel.

This function is also available in the template context as filter named `timedeltaformat`.

## Gettext 函数

`flask.ext.babel.gettext(string, **variables)`

Translates a string with the current locale and passes in the given keyword arguments as mapping to a string formatting string.

```
gettext(u'Hello World!')
gettext(u'Hello %(name)s!', name='World')
```

`flask.ext.babel.ngettext(singular, plural, num, **variables)`

Translates a string with the current locale and passes in the given keyword arguments as mapping to a string formatting string. The `num` parameter is used to dispatch between singular and various plural forms of the message. It is available in the format string as `%(num)d` or `%(num)s`. The source language should be English or a similar language which only has one plural form.

```
ngettext(u'%(num)d Apple', u'%(num)d Apples', num=len(apples))
```

`flask.ext.babel.pgettext(context, string, **variables)`

Like `gettext()` but with a context.

New in version 0.7.

`flask.ext.babel.npgettext(context, singular, plural, num, **variables)`

Like `ngettext()` but with a context.

New in version 0.7.

`flask.ext.babel.lazy_gettext(string, **variables)`

Like `gettext()` but the string returned is lazy which means it will be translated when it is used as an actual string.

Example:

```
hello = lazy_gettext(u'Hello World')

@app.route('/')
def index():
    return unicode(hello)
```

`flask.ext.babel.lazy_pgettext(context, string, **variables)`

Like `pgettext()` but the string returned is lazy which means it will be translated when it is used as an actual string.

New in version 0.7.

## 低层的 API

`flask.ext.babel.refresh()`

Refreshes the cached timezones and locale information. This can be used to switch a translation between a request and if you want the changes to take place immediately, not just with the next request:

```
user.timezone = request.form['timezone']
user.locale = request.form['locale']
refresh()
flash(gettext('Language was changed'))
```

Without that refresh, the `flash()` function would probably return English text and a now German page.

# Flask-Cache

## 安装

使用下面的命令行安装 Flask-Cache:

```
$ easy_install Flask-Cache
```

或者可以用下面的命令行，如果安装了 pip:

```
$ pip install Flask-Cache
```

## 使用

缓存（Cache）是通过使用一个 `Cache` 实例进行管理:

```
from flask import Flask
from flask.ext.cache import Cache

app = Flask(__name__)
# Check Configuring Flask-Cache section for more details
cache = Cache(app,config={'CACHE_TYPE': 'simple'})
```

你能够用 **init_app** 方法在初始化 `Cache` 后设置它:

```
cache = Cache(config={'CACHE_TYPE': 'simple'})

app = Flask(__name__)
cache.init_app(app)
```

如果有多个 `Cache` 实例以及每一个实例都有不同后端的话（换句话说，就是每一个实例使用不用的缓存类型CACHE_TYPE），使用配置字典是十分有用的:

```
#: Method A: During instantiation of class
cache = Cache(config={'CACHE_TYPE': 'simple'})
#: Method B: During init_app call
cache.init_app(app, config={'CACHE_TYPE': 'simple'})
```

New in version 0.7.

## 缓存视图函数

使用装饰器 `cached()` 能够缓存视图函数。它在默认情况下使用请求路径(request.path)作为cache_key:

```
@cache.cached(timeout=50)
def index():
    return render_template('index.html')
```

该装饰器有一个可选的参数：`unless`，它允许一个可调用的、返回值是True或者False的函数。如果 `unless` 返回 `True`，将会完全忽略缓存机制（内置的缓存机制会完全不起作用）。

## 缓存其它函数

同样地，使用 `@cached` 装饰器也能够缓存其它非视图函数的结果。唯一的要求是需要指定 `key_prefix`，否则会使用请求路径(request.path)作为cache_key:

```
@cache.cached(timeout=50, key_prefix='all_comments')
def get_all_comments():
    comments = do_serious_dbio()
    return [x.author for x in comments]

cached_comments = get_all_comments()
```

## Memoization（一种缓存技术）

请参看 `memoize()`

在memoization中，函数参数同样包含cache_key。

Note

如果函数不接受参数的话，`cached()` 和 `memoize()` 两者的作用是一样的。

Memoize同样也为类成员函数而设计，因为它根据 identity 将 ‘self’ 或者 ‘cls’ 参数考虑进作为缓存键的一部分。

memoization背后的理论是：在一次请求中如果一个函数需要被调用多次，它只会计算第一次使用这些参数调用该函数。例如，存在一个决定用户角色的 sqlalchemy对象，在一个请求中可能需要多次调用这个函数。为了避免每次都从数据库获取信息，你可以这样做:

```
class Person(db.Model):
    @cache.memoize(50)
    def has_membership(self, role_id):
            return Group.query.filter_by(user=self, role_id=role_id).count() >= 1
```

Warning

使用可变对象（例如类）作为缓存键的一部分是十分棘手的。建议最好不要让一个对象的实例成为一个memoized函数。然而，memoize在处理参数的时候会执行repr()，因此如果一个对象有 `__repr__` 函数，并且返回一个唯一标识该对象的字符串，它将能够作为缓存键的一部分。

例如，一个sqlalchemy person对象，它返回数据库的ID作为唯一标识符的一部分:

```
class Person(db.Model):
    def __repr__(self):
        return "%s(%s)" % (self.__class__.__name__, self.id)
```

## 删除memoize的缓存

New in version 0.2.

在每个函数的基础上，您可能需要删除缓存。使用上面的例子，让我们来改变用户权限，并将它们分配到一个角色，如果它们新拥有或者失去某些成员关系，现在你需要重新计算。你能够用 `delete_memoized()` 函数来达到目的:

```
cache.delete_memoized('user_has_membership')
```

Note

如果仅仅只有函数名作为参数，所有的memoized的版本将会无效的。然而，您可以删除特定的缓存提供缓存时相同的参数值。在下面的例子中，只有 `user` 角色缓存被删除：

```
user_has_membership('demo', 'admin')
user_has_membership('demo', 'user')

cache.delete_memoized('user_has_membership', 'demo', 'user')
```

# 缓存Jinja2片段

用法:

```
{% cache [timeout [,[key1, [key2, ...]]]] %}
...
{% endcache %}
```

默认情况下“模版文件路径”+“片段开始的函数”用来作为缓存键。同样键名是可以手动设置的。键名串联成一个字符串，这样能够用于避免同样的块在不同模版被重复计算。

设置 timeout 为 None，并且使用了自定义的键:

```
{% cache None "key" %}...
```

为了删除缓存值，为“del”设置超时时间:

```
{% cache 'del' %}...
```

如果提供键名，你可以很容易地产生模版的片段密钥，从模板上下文外删除它:

```
from flask.ext.cache import make_template_fragment_key
key = make_template_fragment_key("key1", vary_on=["key2", "key3"])
cache.delete(key)
```

例子:

```
Considering we have render_form_field and render_submit macroses.
{% cache 60*5 %}

    <form>
    {% render_form_field form.username %}
    {% render_submit %}
    </form>

{% endcache %}
```

# 清除缓存

请参看 `clear()` .

下面的例子是一个用来清空应用缓存的脚本：

```
from flask.ext.cache import Cache

from yourapp import app, your_cache_config

cache = Cache()

def main():
    cache.init_app(app, config=your_cache_config)

    with app.app_context():
        cache.clear()

if __name__ == '__main__':
    main()
```

Warning

某些缓存类型不支持完全清空缓存。同样，如果你不使用键前缀，一些缓存类型将刷新整个数据库。请确保你没有任何其他数据存储在缓存数据库中。

# 配置Flask-Cache

Flask-Cache有下面一些配置项:

`CACHE_TYPE`

指定哪些类型的缓存对象来使用。 这是一个输入字符串，将被导入并实例化。 它假设被导入的对象是一个依赖于werkzeug缓存API， 返回缓存对象的函数。

对于werkzeug.contrib.cache对象，不必给出完整的字符串， 只要是下列这些名称之一。

内建缓存类型：

- **null**: NullCache (default)
- **simple**: SimpleCache
- **memcached**: MemcachedCache (pylibmc or memcache required)
- **gaememcached**: GAEMemcachedCache
- **redis**: RedisCache (Werkzeug 0.7 required)
- **filesystem**: FileSystemCache
- **saslmemcached**: SASLMemcachedCache (pylibmc required)

| | |
|---|---|
| `CACHE_NO_NULL_WARNING` | 当使用的缓存类型是'null'，不会抛出警告信息。 |
| `CACHE_ARGS` | 可选的列表，在缓存类实例化的时候会对该列表进行拆分以及传递（传参）。 |
| `CACHE_OPTIONS` | 可选的字典，在缓存类实例化的时候会传递该字典（传参）。 |
| `CACHE_DEFAULT_TIMEOUT` | 如果没有设置延迟时间，默认的延时时间会被使用。单位为秒。 |
| `CACHE_THRESHOLD` | 最大的缓存条目数，超过该数会删除一些缓存条目。仅仅用于SimpleCache和 FileSystemCache。 |
| `CACHE_KEY_PREFIX` | 所有键之前添加的前缀。 这使得它可以为不同的应用程序使用相同的memcached服务器。 仅仅用于RedisCache，MemcachedCache以及GAEMemcachedCache。 |
| `CACHE_MEMCACHED_SERVERS` | 服务器地址列表或元组。仅用于MemcachedCache。 |
| `CACHE_MEMCACHED_USERNAME` | SASL与memcached服务器认证的用户名。 仅用于SASLMemcachedCache。 |
| `CACHE_MEMCACHED_PASSWORD` | SASL与memcached服务器认证的密码。 仅用于SASLMemcachedCache。 |
| `CACHE_REDIS_HOST` | Redis服务器的主机。仅用于RedisCache。 |
| `CACHE_REDIS_PORT` | Redis服务器的端口。默认是6379。仅用于RedisCache。 |
| `CACHE_REDIS_PASSWORD` | 用于Redis服务器的密码。仅用于RedisCache。 |
| `CACHE_REDIS_DB` | Redis的db库 (基于零号索引)。默认是0。仅用于RedisCache。 |
| `CACHE_DIR` | 存储缓存的目录。仅用于FileSystemCache。 |
| `CACHE_REDIS_URL` | 连接到Redis服务器的URL。 例如：`redis://user:password@localhost:6379/2`。 仅用于RedisCache。 |

此外，如果标准的Flask配置项 `TESTING` 使用并且设置为True的话， **Flask-Cache** 将只会使用NullCache作为缓存类型。

# 内建的缓存类型

## NullCache – null

不缓存内容

- CACHE_ARGS
- CACHE_OPTIONS

## SimpleCache – simple

使用本地Python字典缓存。这不是真正的线程安全。

相关配置

- CACHE_DEFAULT_TIMEOUT
- CACHE_THRESHOLD
- CACHE_ARGS
- CACHE_OPTIONS

## FileSystemCache – filesystem

使用文件系统来存储缓存值

- CACHE_DEFAULT_TIMEOUT
- CACHE_DIR
- CACHE_THRESHOLD
- CACHE_ARGS
- CACHE_OPTIONS

## MemcachedCache – memcached

使用memcached服务器作为后端。支持pylibmc或memcache或谷歌应用程序引擎的memcache库。

相关配置项

- CACHE_DEFAULT_TIMEOUT
- CACHE_KEY_PREFIX
- CACHE_MEMCACHED_SERVERS
- CACHE_ARGS
- CACHE_OPTIONS

## GAEMemcachedCache – gaememcached

MemcachedCache一个不同的名称

## SASLMemcachedCache – saslmemcached

使用memcached服务器作为后端。使用SASL建立与memcached服务器的连接。pylibmc是必须的，libmemcached必须支持SASL。

相关配置项

- CACHE_DEFAULT_TIMEOUT
- CACHE_KEY_PREFIX
- CACHE_MEMCACHED_SERVERS
- CACHE_MEMCACHED_USERNAME
- CACHE_MEMCACHED_PASSWORD
- CACHE_ARGS
- CACHE_OPTIONS

New in version 0.10.

## SpreadSASLMemcachedCache – spreadsaslmemcachedcache

与SASLMemcachedCache一样，但是如果大于memcached的传输安全性，默认是1M，能够跨不同的键名缓存值。使用pickle模块。

New in version 0.11.

## RedisCache – redis

- CACHE_DEFAULT_TIMEOUT
- CACHE_KEY_PREFIX
- CACHE_REDIS_HOST
- CACHE_REDIS_PORT
- CACHE_REDIS_PASSWORD
- CACHE_REDIS_DB
- CACHE_ARGS
- CACHE_OPTIONS
- CACHE_REDIS_URL

# 定制缓存后端（后台）

你能够轻易地定制缓存后端，只需要导入一个能够实例化以及返回缓存对象的函数。 `CACHE_TYPE` 将是你自定义的函数名的字符串。 这个函数期望得到三个参数。

- `app`
- `args`
- `kwargs`

你自定义的缓存对象必须是 `werkzeug.contrib.cache.BaseCache` 的子类。确保 `threshold` 是包含在kwargs参数中，因为它是所有BaseCache类通用的。

Redis的缓存实现的一个例子:

```
#: the_app/custom.py
class RedisCache(BaseCache):
    def __init__(self, servers, default_timeout=500):
        pass

def redis(app, config, args, kwargs):
   args.append(app.config['REDIS_SERVERS'])
   return RedisCache(*args, **kwargs)
```

在这个例子中， `CACHE_TYPE` 可能就是 `the_app.custom.redis` 。

PylibMC缓存实现的一个例子:

```
#: the_app/custom.py
def pylibmccache(app, config, args, kwargs):
    return pylibmc.Client(servers=config['CACHE_MEMCACHED_SERVERS'],
                          username=config['CACHE_MEMCACHED_USERNAME'],
                          password=config['CACHE_MEMCACHED_PASSWORD'],
                          binary=True)
```

在这个例子中， `CACHE_TYPE` 可能就是 `the_app.custom.pylibmccache` 。

# API

`class flask.ext.cache.Cache(app=None, with_jinja2_ext=True, config=None)`

This class is used to control the cache objects.

`add(*args, **kwargs)`

Proxy function for internal cache object.

`cached(timeout=None, key_prefix='view/%s', unless=None)`

Decorator. Use this to cache a function. By default the cache key is `view/request.path` . You are able to use this decorator with any function by changing the `key_prefix` . If the token `%s` is located within the `key_prefix` then it will replace that with `request.path`

Example:

```
# An example view function
@cache.cached(timeout=50)
def big_foo():
    return big_bar_calc()

# An example misc function to cache.
@cache.cached(key_prefix='MyCachedList')
def get_list():
    return [random.randrange(0, 1) for i in range(50000)]

my_list = get_list()
```

Note

You MUST have a request context to actually called any functions that are cached.

New in version 0.4: The returned decorated function now has three function attributes assigned to it. These attributes are readable/writable.

**uncached**

The original undecorated function

**cache_timeout**

The cache timeout value for this function. For a custom value to take affect, this must be set before the function is called.

**make_cache_key**

A function used in generating the cache_key used.

Parameters:

- **timeout** – Default None. If set to an integer, will cache for that amount of time. Unit of time is in seconds.
- **key_prefix** –

  Default 'view/%(request.path)s'. Beginning key to . use for the cache key.

  ```
  New in version 0.3.4: Can optionally be a callable which takes no arguments but retur
  ```

  - **unless** – Default None. Cache will *always* execute the caching facilities unless this callable is true. This will bypass the caching entirely.

`clear()`

Proxy function for internal cache object.

`delete(*args, **kwargs)`

Proxy function for internal cache object.

`delete_many(*args, **kwargs)`

Proxy function for internal cache object.

`delete_memoized(f, *args, **kwargs)`

Deletes the specified functions caches, based by given parameters. If parameters are given, only the functions that were memoized with them will be erased. Otherwise all versions of the caches will be forgotten.

Example:

```
@cache.memoize(50)
def random_func():
    return random.randrange(1, 50)

@cache.memoize()
def param_func(a, b):
    return a+b+random.randrange(1, 50)
```

```
>>> random_func()
43
>>> random_func()
43
>>> cache.delete_memoized('random_func')
>>> random_func()
16
>>> param_func(1, 2)
32
>>> param_func(1, 2)
32
>>> param_func(2, 2)
47
>>> cache.delete_memoized('param_func', 1, 2)
>>> param_func(1, 2)
13
>>> param_func(2, 2)
47
```

Delete memoized is also smart about instance methods vs class methods.

When passing a instancemethod, it will only clear the cache related to that instance of that object. (object uniqueness can be overridden

> When passing a classmethod, it will clear all caches related across all instances of that class.

Example:

```
class Adder(object):
    @cache.memoize()
    def add(self, b):
        return b + random.random()
```

```
>>> adder1 = Adder()
>>> adder2 = Adder()
>>> adder1.add(3)
3.23214234
>>> adder2.add(3)
3.60898509
>>> cache.delete_memoized(adder.add)
>>> adder1.add(3)
3.01348673
>>> adder2.add(3)
3.60898509
>>> cache.delete_memoized(Adder.add)
>>> adder1.add(3)
3.53235667
>>> adder2.add(3)
3.72341788
```

Parameters:

- **fname** – Name of the memoized function, or a reference to the function.
- ***args** – A list of positional parameters used with memoized function.
- ****kwargs** – A dict of named parameters used with memoized function.

Note

Flask-Cache uses inspect to order kwargs into positional args when the function is memoized. If you pass a function reference into `fname` instead of the function name, Flask-Cache will be able to place the args/kwargs in the proper order, and delete the positional cache.

However, if `delete_memoized` is just called with the name of the function, be sure to pass in potential arguments in the same order as defined in your function as args only, otherwise Flask-Cache will not be able to compute the same cache key.

Note

Flask-Cache maintains an internal random version hash for the function. Using delete_memoized will only swap out the version hash, causing the memoize function to recompute results and put them into another key.

This leaves any computed caches for this memoized function within the caching backend.

It is recommended to use a very high timeout with memoize if using this function, so that when the version has is swapped, the old cached results would eventually be reclaimed by the caching backend.

`delete_memoized_verhash(f, *args)`

Delete the version hash associated with the function.

..warning:

```
Performing this operation could leave keys behind that have
been created with this version hash. It is up to the application
to make sure that all keys that may have been created with this
version hash at least have timeouts so they will not sit orphaned
in the cache backend.
```

`get(*args, **kwargs)`

Proxy function for internal cache object.

`get_many(*args, **kwargs)`

Proxy function for internal cache object.

`init_app(app, config=None)`

This is used to initialize cache with your app object

`memoize(timeout=None, make_name=None, unless=None)`

Use this to cache the result of a function, taking its arguments into account in the cache key.

Information on Memoization.

Example:

```
@cache.memoize(timeout=50)
def big_foo(a, b):
    return a + b + random.randrange(0, 1000)
```

```
>>> big_foo(5, 2)
753
>>> big_foo(5, 3)
234
>>> big_foo(5, 2)
753
```

New in version 0.4: The returned decorated function now has three function attributes assigned to it.

**uncached**

The original undecorated function. readable only

**cache_timeout**

The cache timeout value for this function. For a custom value to take affect, this must be set before the function is called.

readable and writable

**make_cache_key**

A function used in generating the cache_key used.

readable and writable

Parameters:

- **timeout** – Default None. If set to an integer, will cache for that amount of time. Unit of time is in seconds.
- **make_name** – Default None. If set this is a function that accepts a single argument, the function name, and returns a new string to be used as the function name. If not set then the function name is used.
- **unless** – Default None. Cache will *always* execute the caching facilities unelss this callable is true. This will bypass the caching entirely.

New in version 0.5: params `make_name` , `unless`

`set(*args, **kwargs)`

Proxy function for internal cache object.

`set_many(*args, **kwargs)`

Proxy function for internal cache object.

# Changelog

## Version 0.13 2014-04-21

- Port to Python >= 3.3 (requiring Python 2.6/2.7 for 2.x).
- Fixed bug with using per-memoize timeouts greater than the default timeout
- Added better support for per-instance memoization.
- Various bug fixes

## Version 0.12 2013-04-29

- Changes jinja2 cache templates to use stable predictable keys. Previously the key for a cache tag included the line number of the template, which made it difficult to predict what the key would be outside of the application.
- Adds config variable `CACHE_NO_NULL_WARNING` to silence warning messages when using 'null' cache as part of testing.
- Adds passthrough to clear entire cache backend.

## Version 0.11.1 2013-04-7

- Bugfix for using memoize on instance methods. The previous key was id(self), the new key is repr(self)

## Version 0.11 2013-03-23

- Fail gracefully in production if cache backend raises an exception.
- Support for redis DB number
- Jinja2 templatetag cache now concats all args together into a single key instead of treating each arg as a separate key name.
- Added delete memcache version hash function
- Support for multiple cache objects on a single app again.
- Added SpreadSASLMemcached, if a value is greater than the memcached threshold which defaults to 1MB, this splits the value across multiple keys.
- Added support to use URL to connect to redis.

## Version 0.10.1 2013-01-13

- Added warning message when using cache type of ‘null’
- Changed imports to relative instead of absolute for AppEngine compatibility

## Version 0.10.0 2013-01-05

- Added `saslmemcached` backend to support Memcached behind SASL authentication.
- Fixes a bug with memoize when the number of args != number of kwargs

## Version 0.9.2 2012-11-18

- Bugfix with default kwargs

## Version 0.9.1 2012-11-16

- Fixes broken memoized on functions that use default kwargs

## Version 0.9.0 2012-10-14

- Fixes memoization to work on methods.

## Version 0.8.0 2012-09-30

- Migrated to the new flask extension naming convention of flask_cache instead of flaskext.cache

- Removed unnecessary dependencies in setup.py file.
- Documentation updates

## Version 0.7.0 2012-08-25

- Allows multiple cache objects to be instantiated with different configuration values.

## Version 0.6.0 2012-08-12

- Memoization is now safer for multiple applications using the same backing store.
- Removed the explicit set of NullCache if the Flask app is set testing=True
- Swapped Conditional order for key_prefix

## Version 0.5.0 2012-02-03

- Deleting memoized functions now properly functions in production environments where multiple instances of the application are running.
- get_memoized_names and get_memoized_keys have been removed.
- Added `make_name` to memoize, make_name is an optional callable that can be passed to memoize to modify the cache_key that gets generated.
- Added `unless` to memoize, this is the same as the unless parameter in `cached`
- memoization now converts all kwargs to positional arguments, this is so that when a function is called multiple ways, it would evaluate to the same cache_key

## Version 0.4.0 2011-12-11

- Added attributes for uncached, make_cache_key, cache_timeout to the decorated functions.

## Version 0.3.4 2011-09-10

- UTF-8 encoding of cache key
- key_prefix argument of the cached decorator now supports callables.

## Version 0.3.3 2011-06-03

Uses base64 for memoize caching. This fixes rare issues where the cache_key was either a tuple or larger than the caching backend would be able to support.

Adds support for deleting memoized caches optionally based on function parameters.

Python 2.5 compatibility, plus bugfix with string.format.

Added the ability to retrieve memoized function names or cache keys.

## Version 0.3.2

Bugfix release. Fixes a bug that would cause an exception if no `CACHE_TYPE` was supplied.

## Version 0.3.1

Pypi egg fix.

## Version 0.3

- CACHE_TYPE changed. Now one of [‘null’, ‘simple’, ‘memcached’, ‘gaememcached’, ‘filesystem’], or an import string to a function that will instantiate a cache object. This allows Flask-Cache to be much more extensible and configurable.

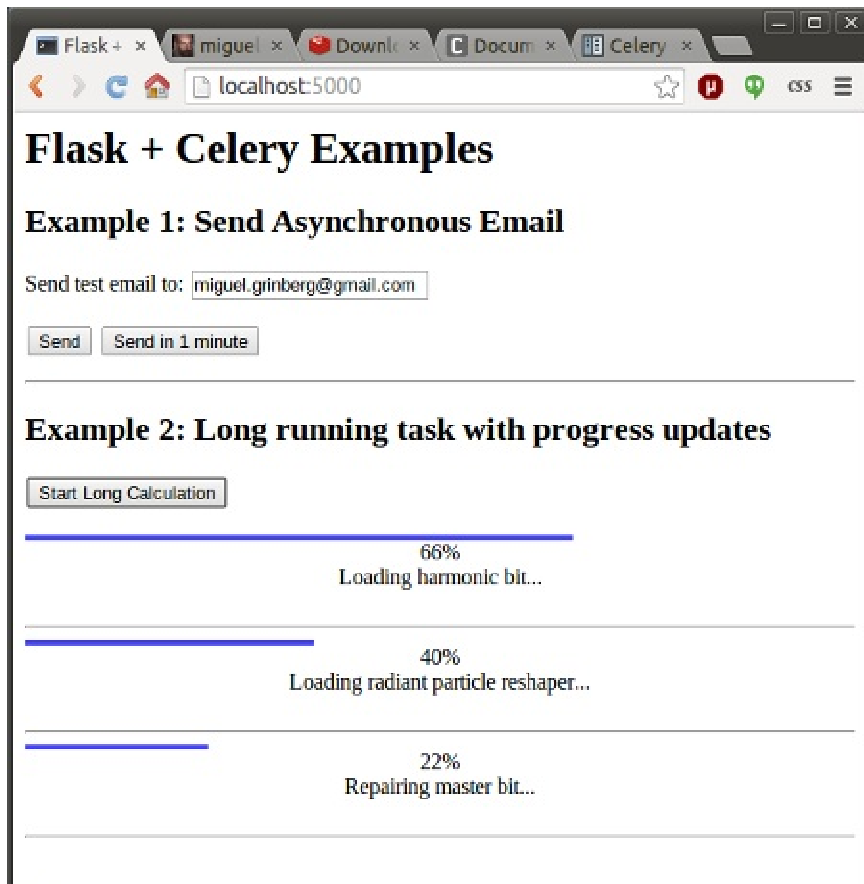
## Version 0.2

- CACHE_TYPE now uses an import_string.
- Added CACHE_OPTIONS and CACHE_ARGS configuration values.
- Added delete_memoized

## Version 0.1

- Initial public release

# 在 Flask 中使用 Celery




后台运行任务的话题是有些复杂，因为围绕这个话题会让人产生困惑。为了简单起见，在以前我所有的例子中，我都是在线程中执行后台任务，但是我一直注意到更具有扩展性以及具备生产解决方案的任务队列像 Celery 应该可以替代线程中执行后台任务。

不断有读者问我关于 Celery 问题，以及怎样在 Flask 应用中使用它，因此今天我将会向你们展示两个例子，我希望能够覆盖大部分的应用需求。

# 什么是 Celery?

Celery 是一个异步任务队列。你可以使用它在你的应用上下文之外执行任务。总的想法就是你的应用程序可能需要执行任何消耗资源的任务都可以交给任务队列，让你的应用程序自由和快速地响应客户端请求。

使用 Celery 运行后台任务并不像在线程中这样做那么简单。但是好处多多，Celery 具有分布式架构，使你的应用易于扩展。一个 Celery 安装有三个核心组件：

1. Celery 客户端: 用于发布后台作业。当与 Flask 一起工作的时候，客户端与 Flask 应用一起运行。

2. Celery workers: 这些是运行后台作业的进程。Celery 支持本地和远程的 workers，因此你就可以在 Flask 服务器上启动一个单独的 worker，随后随着你的应用需求的增加而新增更多的 workers。
3. 消息代理: 客户端通过消息队列和 workers 进行通信，Celery 支持多种方式来实现这些队列。最常用的代理就是 RabbitMQ 和 Redis。

## 致行动派读者

如果你是行动派，本文开头的截图勾起你的好奇心的话，那么可以直接到 Github repository 获取本文用到的代码。 README 文件将会给你快速和直接的方式去运行示例应用。

接着可以回到本文来了解工作机制！

## Flask 和 Celery 一起工作

Flask 与 Celery 整合是十分简单，不需要任何插件。一个 Flask 应用需要使用 Celery 的话只需要初始化 Celery 客户端像这样:

```
from flask import Flask
from celery import Celery

app = Flask(__name__)
app.config['CELERY_BROKER_URL'] = 'redis://localhost:6379/0'
app.config['CELERY_RESULT_BACKEND'] = 'redis://localhost:6379/0'

celery = Celery(app.name, broker=app.config['CELERY_BROKER_URL'])
celery.conf.update(app.config)
```

正如你所见，Celery 通过创建一个 Celery 类对象来初始化，传入应用名称以及消息代理的连接 URL，这个 URL 我把它放在 app.config 中的 CELERY_BROKER_URL 的键值。URL 告诉 Celery 代理服务在哪里运行。如果你运行的不是 Redis，或者代理服务运行在一个不同的机器上，相应地你需要改变 URL。

Celery 其它任何配置可以直接用 celery.conf.update() 通过 Flask 的配置直接传递。CELERY_RESULT_BACKEND 选项只有在你必须要 Celery 任务的存储状态和运行结果的时候才是必须的。展示的第一个示例是不需要这个功能的，但是第二个示例是需要的，因此最好从一开始就配置好。

任何你需要作为后台任务的函数需要用 celery.task 装饰器装饰。例如:

```
@celery.task
def my_background_task(arg1, arg2):
    # some long running task here
    return result
```

接着 Flask 应用能够请求这个后台任务的执行，像这样:

```
task = my_background_task.delay(10, 20)
```

delay() 方法是强大的 apply_async() 调用的快捷方式。这样相当于使用 apply_async():

```
task = my_background_task.apply_async(args=[10, 20])
```

当使用 apply_async()，你可以给 Celery 后台任务如何执行的更详细的说明。一个有用的选项就是要求任务在未来的某一时刻执行。例如，这个调用将安排任务运行在大约一分钟后:

```
task = my_background_task.apply_async(args=[10, 20], countdown=60)
```

delay() 和 apply_async() 的返回值是一个表示任务的对象，这个对象可以用于获取任务状态。我将会在本文的后面展示如何获取任务状态等信息，但现在让我们保持简单些，不用担心任务的执行结果。

更多可用的选项请参阅 Celery 文档 。

# 简单例子：异步发送邮件

我要举的第一个示例是应用程序非常普通的需求：能够发送邮件但是不阻塞主应用。

在这个例子中我会用到 Flask-Mail 扩展，我会假设你们熟悉这个扩展。

我用来说明的示例应用是一个只有一个输入文本框的简单表单。要求用户在此文本框中输入一个电子邮件地址，并在提交，服务器会发送一个测试电子邮件到这个邮件地址。表单中包含两个提交按钮，一个立即发送邮件，一个是一分钟后发送邮件。表单的截图在文章开始。

这里就是支持这个示例的 HTML 模板:

```
<html>
  <head>
    <title>Flask + Celery Examples</title>
  </head>
  <body>
    <h1>Flask + Celery Examples</h1>
    <h2>Example 1: Send Asynchronous Email</h2>
    {% for message in get_flashed_messages() %}
    <p style="color: red;">{{ message }}</p>
    {% endfor %}
    <form method="POST">
      <p>Send test email to: <input type="text" name="email" value="{{ email }}"></p>
      <input type="submit" name="submit" value="Send">
      <input type="submit" name="submit" value="Send in 1 minute">
    </form>
  </body>
</html>
```

这里没有什么特别的东西。只是一个普通的 HTML 表单，再加上 Flask 闪现消息。

Flask-Mail 扩展需要一些配置，尤其是电子邮件服务器发送邮件的时候会用到一些细节。为了简单我使用我的 Gmail 账号作为邮件服务器:

```
# Flask-Mail configuration
app.config['MAIL_SERVER'] = 'smtp.googlemail.com'
app.config['MAIL_PORT'] = 587
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = os.environ.get('MAIL_USERNAME')
app.config['MAIL_PASSWORD'] = os.environ.get('MAIL_PASSWORD')
app.config['MAIL_DEFAULT_SENDER'] = 'flask@example.com'
```

注意为了避免我的账号丢失的风险，我将其设置在系统的环境变量，这是我从应用中导入的。

有一个单一的路由来支持这个示例:

```
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'GET':
        return render_template('index.html', email=session.get('email', ''))
    email = request.form['email']
    session['email'] = email

    # send the email
    msg = Message('Hello from Flask',
                  recipients=[request.form['email']])
    msg.body = 'This is a test email sent from a background Celery task.'
    if request.form['submit'] == 'Send':
        # send right away
        send_async_email.delay(msg)
        flash('Sending email to {0}'.format(email))
    else:
        # send in one minute
        send_async_email.apply_async(args=[msg], countdown=60)
        flash('An email will be sent to {0} in one minute'.format(email))

    return redirect(url_for('index'))
```

再次说明，这是一个很标准的 Flask 应用。由于这是一个非常简单的表单，我决定在没有扩展的帮助下处理它，因此我用 request.method 和 request.form 来完成所有的管理。我保存用户在文本框中输入的值在 session 中，这样在页面重新加载后就能记住它。

在这个函数中让人有兴趣的是发送邮件的时候是通过调用一个叫做 send_async_email 的 Celery 任务，该任务调用 delay() 或者 apply_async() 方法。

这个应用的最后一部分就是能够完成作业的异步任务:

```
@celery.task
def send_async_email(msg):
    """Background task to send an email with Flask-Mail."""
    with app.app_context():
        mail.send(msg)
```

这个任务使用 celery.task 装饰使得成为一个后台作业。这个函数唯一值得注意的就是 Flask-Mail 需要在应用的上下文中运行，因此需要在调用 send() 之前创建一个应用上下文。

重点注意在这个示例中从异步调用返回值并不保留，因此应用不能知道调用成功或者失败。当你运行这个示例的时候，需要检查 Celery worker 的输出来排查发送邮件的问题。

# 复杂例子：显示状态更新和结果

上面的示例过于简单，后台作业启动然后应用忘记它。大部分 Celery 针对网页开发的教程就到此为止，但是事实上许多应用程序有必要监控它的后台任务并且获取运行结果。

我现在将要做的就是扩展上面的应用程序成为第二个示例，这个示例展示一个虚构的长时间运行的任务。用户点击按钮启动一个或者更多的长时间运行的任务，在浏览器上的页面使用 ajax 轮询服务器更新所有任务的状态。每一个任务，页面都会显示一个图形的状态栏，进度条，一个状态消息，并且当任务完成的时候，也会显示任务的执行结果。示例的截图在本文的最开始。

## 状态更新的后台任务

让我向你们展示我在第二个示例中使用的后台任务:

```
@celery.task(bind=True)
def long_task(self):
    """Background task that runs a long function with progress reports."""
    verb = ['Starting up', 'Booting', 'Repairing', 'Loading', 'Checking']
    adjective = ['master', 'radiant', 'silent', 'harmonic', 'fast']
    noun = ['solar array', 'particle reshaper', 'cosmic ray', 'orbiter', 'bit']
    message = ''
    total = random.randint(10, 50)
    for i in range(total):
        if not message or random.random() < 0.25:
            message = '{0} {1} {2}...'.format(random.choice(verb),
                                              random.choice(adjective),
                                              random.choice(noun))
        self.update_state(state='PROGRESS',
                          meta={'current': i, 'total': total,
                                'status': message})
        time.sleep(1)
    return {'current': 100, 'total': 100, 'status': 'Task completed!',
            'result': 42}
```

对于这个任务，我在 Celery 装饰器中添加了 bind=True 参数。这个参数告诉 Celery 发送一个 self 参数到我的函数，我能够使用它(self)来记录状态更新。

因为这个任务真没有干什么有用的事情，我决定使用随机的动词，形容词和名词组合的幽默状态信息。你可以在代码上看到我用来生成上述信息的毫无意义的列表。

self.update_state() 调用是 Celery 如何接受这些任务更新。有一些内置的状态，比如 STARTED, SUCCESS 等等，但是 Celery 也支持自定义状态。这里我使用一个叫做 PROGRESS 的自定义状态。连同状态，还有一个附件的元数据，该元数据是 Python 字典形式，包含目前和总的迭代数以及随机生成的状态消息。客户端可以使用这些元素来显示一个漂亮的进度条。每迭代一次休眠一秒，以模拟正在做一些工作。

当循环退出，一个 Python 字典作为函数结果返回。这个字典包含了更新迭代计数器，最后的状态消息和幽默的结果。

上面的 long_task() 函数在一个 Celery worker 进程中运行。下面你能看到启动这个后台作业的 Flask 应用路由:

```
@app.route('/longtask', methods=['POST'])
def longtask():
    task = long_task.apply_async()
    return jsonify({}), 202, {'Location': url_for('taskstatus',
                                              task_id=task.id)}
```

正如你所见，客户端需要发起一个 POST 请求到 /longtask 来掀开这些任务中的一个的序幕。服务器启动任务，并且存储返回值。对于响应我使用状态码 202，这个状态码通常是在 REST APIs 中使用用来表明一个请求正在进行中。我也添加了 Location 头，值为一个客户端用来获取状态信息的 URL。这个 URL 指向另一个叫做 taskstatus 的 Flask 路由，并且有 task.id 作为动态的要素。

## 从 Flask 应用中访问任务状态

上面提及到 taskstatus 路由负责报告有后台任务提供的状态更新。这里就是这个路由的实现:

```
@app.route('/status/<task_id>')
def taskstatus(task_id):
    task = long_task.AsyncResult(task_id)
    if task.state == 'PENDING':
        // job did not start yet
        response = {
            'state': task.state,
            'current': 0,
            'total': 1,
            'status': 'Pending...'
        }
    elif task.state != 'FAILURE':
        response = {
            'state': task.state,
            'current': task.info.get('current', 0),
            'total': task.info.get('total', 1),
            'status': task.info.get('status', '')
        }
        if 'result' in task.info:
            response['result'] = task.info['result']
    else:
        # something went wrong in the background job
        response = {
            'state': task.state,
            'current': 1,
            'total': 1,
            'status': str(task.info),  # this is the exception raised
        }
    return jsonify(response)
```

这个路由生成一个 JSON 响应，该响应包含任务的状态以及设置在 update_state() 调用中作为 meta 的参数的所有值，客户端可以使用这些构建一个进度条。遗憾地是这个函数需要检查一些条件，因此代码有些长。为了能够访问任务的数据，我重新创建了任务对象，该对象是

AsyncResult 类的实例，使用了 URL 中给的任务 id。

第一个 if 代码块是当任务还没有开始的时候(PENDING 状态)。在这种情况下暂时没有状态信息，因此我人为地制造了些数据。接下来的 elif 代码块返回后台的任务的状态信息。任务提供的信息可以通过访问 task.info 获得。如果数据中包含键 result ，这就意味着这是最终的结果并且任务已经结束，因此我把这些信息也加到响应中。最后的 else 代码块是任务执行失败的情况，这种情况下 task.info 中会包含异常的信息。

不管你是否相信，服务器所有要做的事情已经完成了。剩下的部分就是需要客户端需要实现的，在这里也就是用 JavaScript 脚本的网页来实现。

## 客户端的 Javascript

这一部分就不是本文的重点，如果你有兴趣的话，可以自己研究研究。

对于图形进度条我使用 nanobar.js，我从 CDN 上引用它。同样还需要引入 jQuery，它能够简化 ajax 的调用。

```
<script src="//cdnjs.cloudflare.com/ajax/libs/nanobar/0.2.1/nanobar.min.js"></script>
<script src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
```

启动连接后台作业的按钮的 Javascript 处理程序如下:

```
function start_long_task() {
    // add task status elements
    div = $('0%... <hr>');
    $('#progress').append(div);

    // create a progress bar
    var nanobar = new Nanobar({
        bg: '#44f',
        target: div[0].childNodes[0]
    });

    // send ajax POST request to start background job
    $.ajax({
        type: 'POST',
        url: '/longtask',
        success: function(data, status, request) {
            status_url = request.getResponseHeader('Location');
            update_progress(status_url, nanobar, div[0]);
        },
        error: function() {
            alert('Unexpected error');
        }
    });
}
```

div 的代码:

```
            <-- Progress bar
    0%      <-- Percentage
    ...     <-- Status message
        <-- Result

<hr>
```

最后 Javascript 的 update_progress 函数代码如下:

```
function update_progress(status_url, nanobar, status_div) {
      // send GET request to status URL
      $.getJSON(status_url, function(data) {
          // update UI
          percent = parseInt(data['current'] * 100 / data['total']);
          nanobar.go(percent);
          $(status_div.childNodes[1]).text(percent + '%');
          $(status_div.childNodes[2]).text(data['status']);
          if (data['state'] != 'PENDING' && data['state'] != 'PROGRESS') {
              if ('result' in data) {
                  // show result
                  $(status_div.childNodes[3]).text('Result: ' + data['result']);
              }
              else {
                  // something unexpected happened
                  $(status_div.childNodes[3]).text('Result: ' + data['state']);
              }
          }
          else {
              // rerun in 2 seconds
              setTimeout(function() {
                  update_progress(status_url, nanobar, status_div);
              }, 2000);
          }
      });
  }
```

这一部分的代码就不一一解释了。

# 运行示例

首先下载代码，代码的位于 Github repository，接着执行以下的命令:

```
$ git clone https://github.com/miguelgrinberg/flask-celery-example.git
$ cd flask-celery-example
$ virtualenv venv
$ source venv/bin/activate
(venv) $ pip install -r requirements.txt
```

接着，启动 redis，关于 redis 的安装，启动以及配置，请参阅 Redis 文档。

最后，执行如下命令运行示例:

```
$ export MAIL_USERNAME=<your-gmail-username>
$ export MAIL_PASSWORD=<your-gmail-password>
$ source venv/bin/activate
(venv) $ celery worker -A app.celery --loglevel=info
```

运行你的 Flask 应用来感受 Flask 和 Celery 一起工作的快乐:

```
$ source venv/bin/activate
(venv) $ python app.py
```

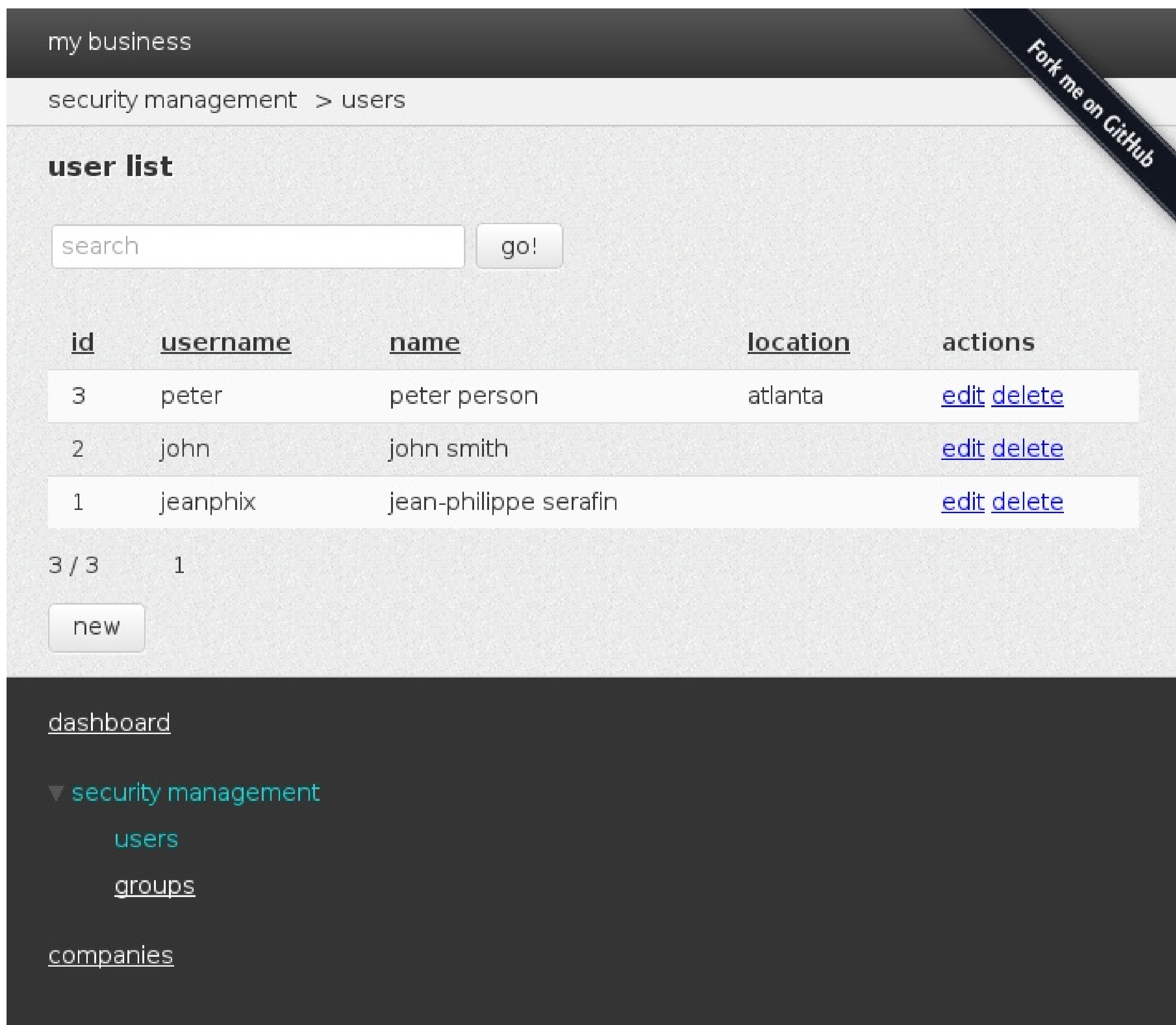
# 欢迎来到 Flask_Dashed 的文档!

## 介绍

Flask-Dashed 提供构建简单以及具有扩展性的管理界面的工具。

在线演示: http://flask-dashed.jeanphi.fr/ (需要 Github 账号)。

列表视图:




表单视图:

# 安装

```
pip install Flask-Dashed
```

# 最小的使用

代码:

```
from flask import Flask
from flask_dashed.admin import Admin

app = Flask(__name__)
admin = Admin(app)

if __name__ == '__main__':
    app.run()
```

示例应用程序: http://github.com/jeanphix/flask-dashed-demo

# 安全地处理

保护("武装")所有模块端:

```
from flask import session

book_module = admin.register_module(BookModule, '/books', 'books',
    'book management')

@book_module.secure(http_code=401)
def login_required():
    return "user" in session
```

保护("武装")特定的模块端:

```
@book_module.secure_endpoint('edit', http_code=403)
def check_edit_credential(view):
    # I'm now signed in, may I modify the ressource?
    return session.user.can_edit_book(view.object)
```

# 模块组织

由于管理节点( node )注册到一个"树",因此很容易地管理它们。:

```
library = admin.register_node('/library', 'library', my library)
book_module = admin.register_module(BookModule, '/books', 'books',
    'book management', parent=library)
```

为了满足你的需求,导航和面包屑会自动地建立。子模块的安全将会继承自父模块。

# SQLALchemy 扩展

Code:

```
from flask_dashed.ext.sqlalchemy import ModelAdminModule

class BookModule(ModelAdminModule):
    model = Book
    db_session = db.session

book_module = admin.register_module(BookModule, '/books', 'books',
    'book management')
```

# Api

# Admin 对象

```
class admin.Admin(app, url_prefix='/admin', title='flask-dashed', main_dashboard=None, end
```

Class that provides a way to add admin interface to Flask applications.

Parameters:

- **app** – The Flask application
- **url_prefix** – The url prefix
- **main_dashboard** – The main dashboard object
- **endpoint** – The endpoint

```
add_path_security(path, function, http_code=403)
```

Registers security function for given path.

Parameters:

- **path** – The endpoint to secure
- **function** – The security function
- **http_code** – The response http code

```
check_path_security(path)
```

Checks security for specific and path.

Parameters: **path** – The path to check

```
register_module(module_class, url_prefix, endpoint, short_title, title=None, parent=None)
```

Registers new module to current admin.

```
register_node(url_prefix, endpoint, short_title, title=None, parent=None, node_class=<clas
```

Registers admin node.

Parameters:

- **url_prefix** – The url prefix
- **endpoint** – The endpoint
- **short_title** – The short title
- **title** – The long title
- **parent** – The parent node path
- **node_class** – The class for node objects

# Admin 模块

```
admin.recursive_getattr(obj, attr)
```

Returns object related attributes, as it’s a template filter None is return when attribute doesn’t exists.

eg:

```
a = object()
a.b = object()
a.b.c = 1
recursive_getattr(a, 'b.c') => 1
recursive_getattr(a, 'b.d') => None
```

`class admin.AdminNode(admin, url_prefix, endpoint, short_title, title=None, parent=None)`

An AdminNode just act as navigation container, it doesn’t provide any rules.

Parameters:

- **admin** – The parent admin object
- **url_prefix** – The url prefix
- **enpoint** – The endpoint
- **short_title** – The short module title use on navigation & breadcrumbs
- **title** – The long title
- **parent** – The parent node

`parents`

Returns all parent hierarchy as list. Usefull for breadcrumbs.

`secure(http_code=403)`

Gives a way to secure specific url path.

Parameters: **http_code** – The response http code when False

`url_path`

Returns the url path relative to admin one.

`class admin.AdminModule(*args, **kwargs)`

Class that provides a way to create simple admin module.

Parameters:

- **admin** – The parent admin object
- **url_prefix** – The url prefix
- **enpoint** – The endpoint
- **short_title** – the short module title use on navigation & breadcrumbs
- **title** – The long title
- **parent** – The parent node

`add_url_rule(rule, endpoint, view_func, **options)`

Adds a routing rule to the application from relative endpoint. `view_class` is copied as we need to dynamically apply decorators.

Parameters:

- **rule** – The rule
- **endpoint** – The endpoint
- **view_func** – The view

`secure_endpoint(endpoint, http_code=403)`

Gives a way to secure specific url path.

Parameters:

- **endpoint** – The endpoint to protect
- **http_code** – The response http code when False

`url`

Returns first registered (main) rule as url.

# SQLAlchemy 扩展

`class ext.sqlalchemy.ModelAdminModule(*args, **kwargs)`

SQLAlchemy model admin module builder.

`count_list(search=None)`

Counts filtered list.

Parameters: **search** – The string for quick search

`create_object()`

New object instance new object.

`delete_object(object)`

Deletes object.

Parameters: **object** – The object to delete

`edit_query_factory`

Returns query for object edition.

`form_view`

alias of `ObjectFormView`

`get_actions_for_object(object)`

“Returns actions for object as and tuple list.

Parameters: **object** – The object

`get_object(pk)`

Gets back object by primary key.

Parameters: **pk** – The object primary key

`get_object_list(search=None, order_by_name=None, order_by_direction=None, offset=None, lim`

Returns ordered, filtered and limited query.

Parameters:

- **search** – The string for search filter
- **order_by_name** – The field name to order by
- **order_by_direction** – The field direction
- **offset** – The offset position
- **limit** – The limit

`list_query_factory`

Returns non filtered list query.

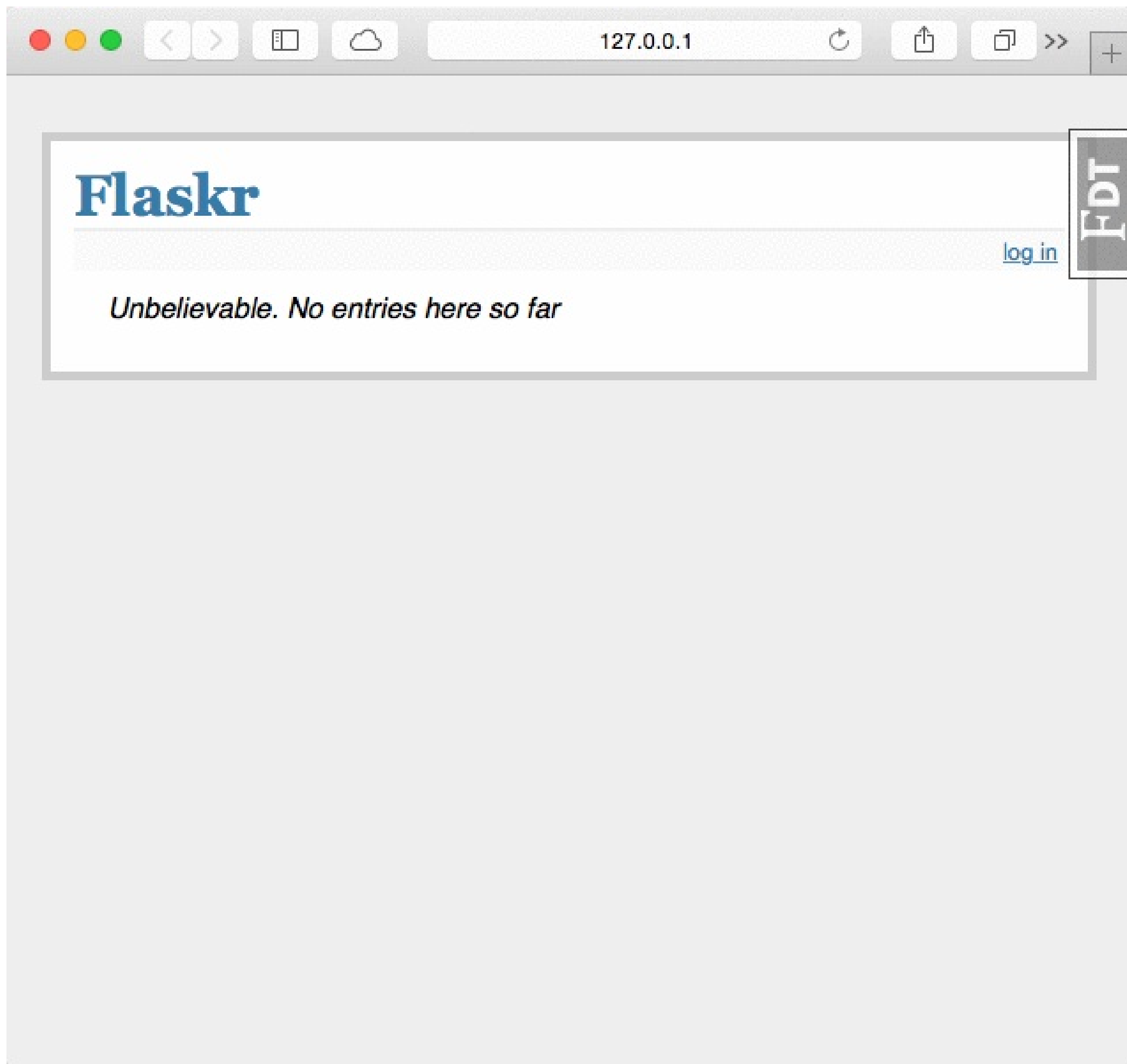
`save_object(obj)`

Saves object.

Parameters: **object** – The object to save

# Flask-DebugToolbar

该扩展为 Flask 应用程序添加了一个包含有用的调试信息的工具栏。




## 安装

简单地使用 pip 来安装:

```
$ pip install flask-debugtoolbar
```

## 用法

设置调试工具栏是简单的:

```
from flask import Flask
from flask_debugtoolbar import DebugToolbarExtension

app = Flask(__name__)

# the toolbar is only enabled in debug mode:
app.debug = True

# set a 'SECRET_KEY' to enable the Flask session cookies
app.config['SECRET_KEY'] = '<replace with a secret key>'

toolbar = DebugToolbarExtension(app)
```

当调试模式开启的时候，工具栏会自动地给添加到 Jinja 模板中。在生产环境中，设置 `app.debug = False` 将会禁用工具栏。

该扩展也支持 Flask 应用的工厂模式，先单独地创建工具栏接着后面为应用初始化它:

```
toolbar = DebugToolbarExtension()
# Then later on.
app = create_app('the-config.cfg')
toolbar.init_app(app)
```

## 配置

工具栏支持多个配置选项:

| 名称 | 描述 | 默认值 |
|---|---|---|
| `DEBUG_TB_ENABLED` | 启用工具栏？ | `app.debug` |
| `DEBUG_TB_HOSTS` | 显示工具栏的 hosts 白名单 | 任意 host |
| `DEBUG_TB_INTERCEPT_REDIRECTS` | 要拦截重定向？ | `True` |
| `DEBUG_TB_PANELS` | 面板的模板/类名的清单 | 允许所有内置的面板 |
| `DEBUG_TB_PROFILER_ENABLED` | 启用所有请求的分析工具 | `False`，用户自行开启 |
| `DEBUG_TB_TEMPLATE_EDITOR_ENABLED` | 启用模板编辑器 | `False` |

要更改配置选项之一，在 Flask 应用程序配置中像这样设置它:

```
app.config['DEBUG_TB_INTERCEPT_REDIRECTS'] = False
```

# 内置的 Panels

## 版本

`flask_debugtoolbar.panels.versions.VersionDebugPanel`

显示已安装的 Flask 版本。展开的视图显示了由 `setuptools` 发现的所有已安装的包和它们的版本。

## 时间

`flask_debugtoolbar.panels.timer.TimerDebugPanel`

显示处理当前请求的时间。展开后的视图包含了用户态和系统态，执行时间，上下文切换的 CPU 时间分解。

Resource Usage

| Resource | Value |
|---|---|
| User CPU time | 12.783 msec |
| System CPU time | 2.289 msec |
| Total CPU time | 15.072 msec |
| Elapsed time | 17.563 msec |
| Context switches | 3 voluntary, 92 involuntary |

Hide »
Versions
Flask 0.10.1
Time
CPU: 15.07ms (17.56ms)
HTTP Headers
Request Vars
Config

## HTTP 头

`flask_debugtoolbar.panels.headers.HeaderDebugPanel`

显示了目前请求的 HTTP 头。

HTTP Headers

| Key | Value |
|---|---|
| CONTENT_TYPE | |
| HTTP_ACCEPT | text/html,application/xhtml+xml,applic |
| HTTP_ACCEPT_ENCODING | gzip, deflate |
| HTTP_ACCEPT_LANGUAGE | en-us |
| HTTP_CONNECTION | keep-alive |
| HTTP_HOST | 127.0.0.1:5000 |
| HTTP_USER_AGENT | Mozilla/5.0 (Macintosh; Intel Mac OS X AppleWebKit/600.2.5 (KHTML, like Gecko) Safari/600.2.5 |
| QUERY_STRING | |

Hide »
Versions
Flask 0.10.1
Time
CPU: 15.07ms (17.56ms)
HTTP Headers
Request Vars
Config

# Request 变量

`flask_debugtoolbar.panels.request_vars.RequestVarsDebugPanel`

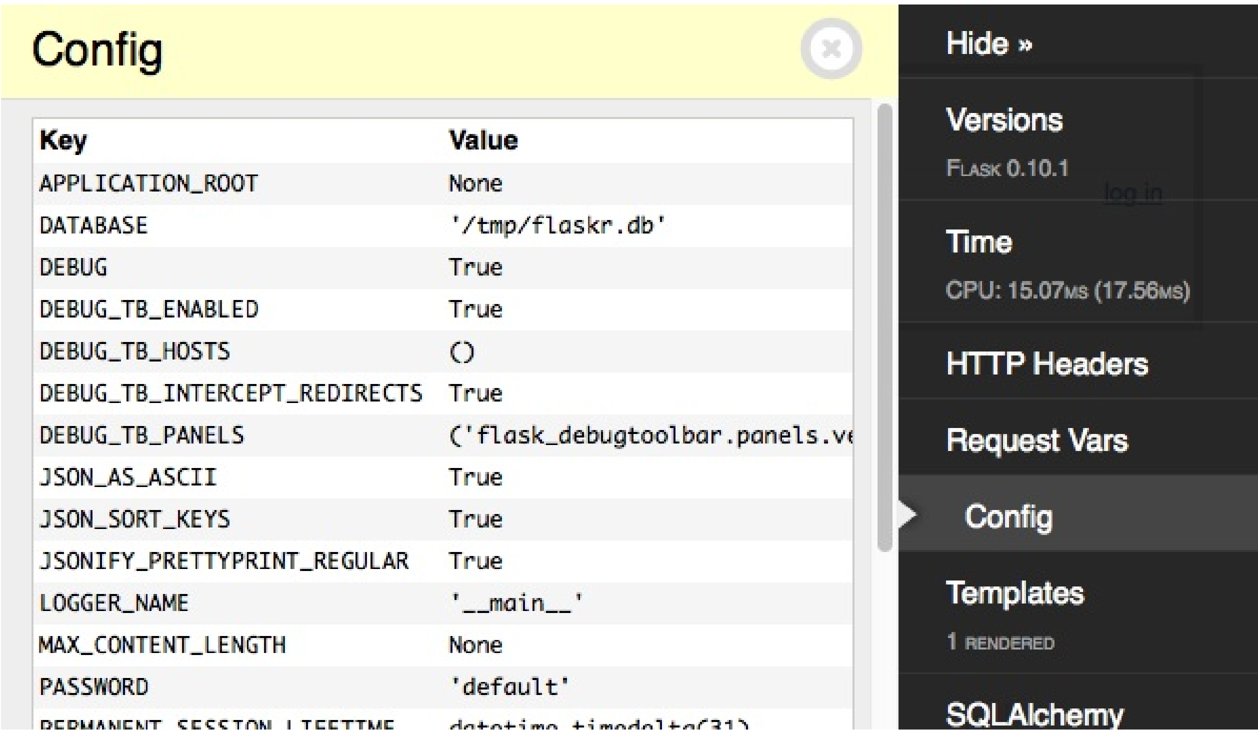
展现了 Flask 请求相关的变量的细节，包含视图函数变量，会话变量，以及 GET 和 POST 变量。



# 配置

`flask_debugtoolbar.panels.config_vars.ConfigVarsDebugPanel`

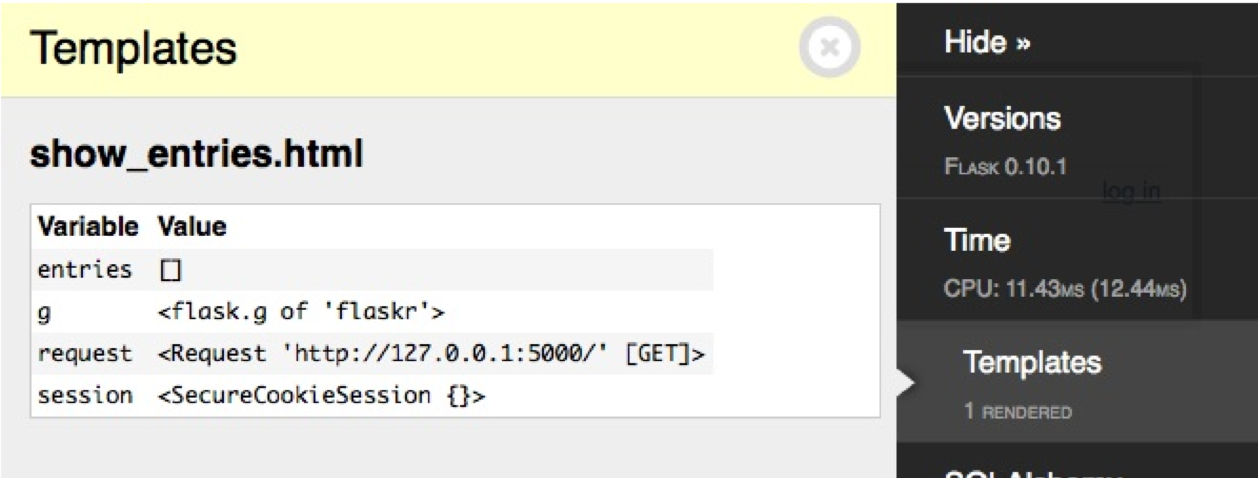
显示了 Flask 应用程序配置字典 `app.config` 的内容。




# 模板

`flask_debugtoolbar.panels.template.TemplateDebugPanel`

显示关于为某个请求渲染模板的信息，以及提供的模板参数的值。




# SQLAlchemy

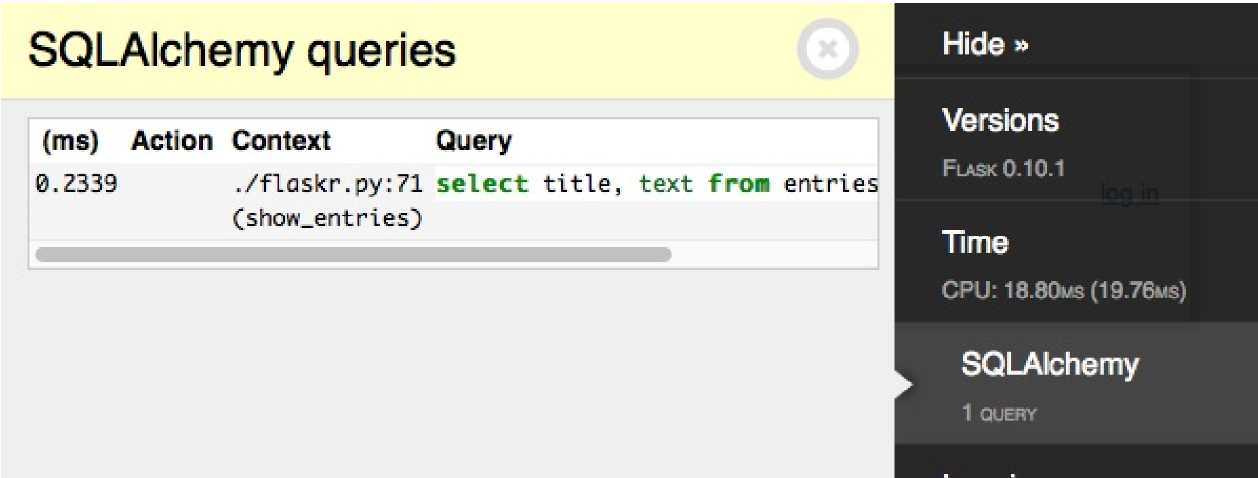
`flask_debugtoolbar.panels.sqlalchemy.SQLAlchemyDebugPanel`

显示了当前请求过程中运行的 SQL 查询。

Note

为了记录查询，这个面板需要使用 Flask-SQLAlchemy 扩展。 请查看 Flask-SQLAlchemy 的 *Quickstart* 章节来配置它。

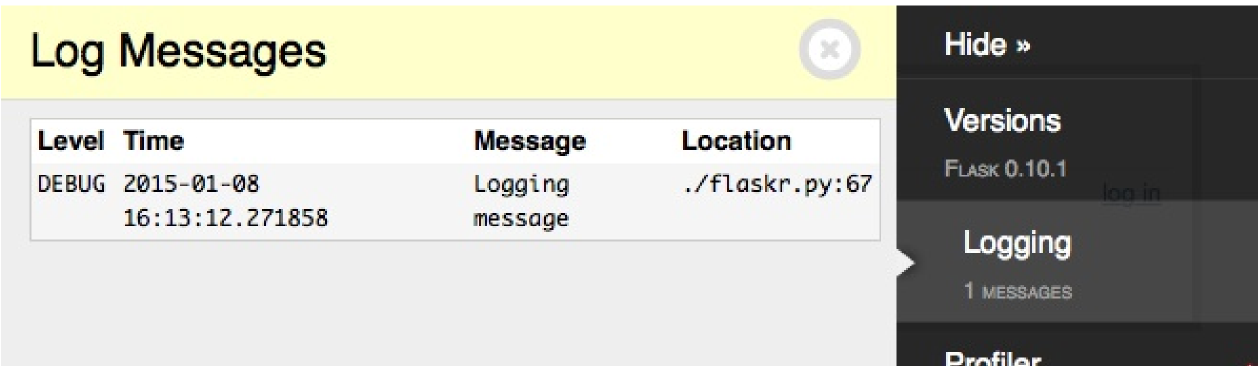
更多关于查询记录的信息请查看文档 `get_debug_queries()` 。




# 日志

`flask_debugtoolbar.panels.logger.LoggingPanel`

显示了当前请求的日志信息




# 路由列表

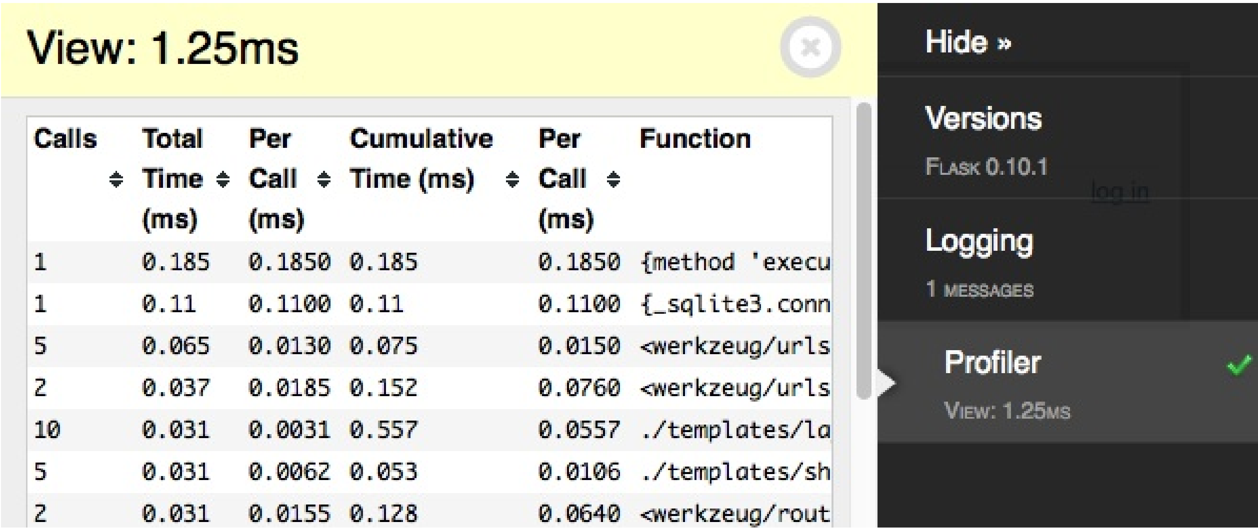
`flask_debugtoolbar.panels.route_list.RouteListDebugPanel`

显示了 Flask URL 路由规则。

# 分析/探查

`flask_debugtoolbar.panels.profiler.ProfilerDebugPanel`

报告当前请求的分析/探查数据。由于性能的考虑，默认情况下分析/探查是禁用的。单击点中选择分析/探查的标记来决定开启或者关闭。在启用分析/探查后，重新刷新页面来运行分析/探查。




# 贡献

Fork 我们在 GitHub 上。

# 感谢

本扩展是基于 django-debug-toolbar 。多谢 Michael van Tellingen 为了这个 Flask 扩展最初的开发，并且感谢 individual contributors 中的每一个人。

# Flask-Exceptional

Flask-Exceptional 为 Flask 添加了 Exceptional 支持。Exceptional 会捕获到你的应用程序中的错误，实时地报告它们，并且会收集你需要快速地修复它们的一些信息。访问 http://www.exceptional.io 去试试。

## 安装

接下来的文档是假设你拥有一个 Exceptional 账号。用 pip 安装这个扩展是简单的:

```
$ pip install Flask-Exceptional
```

或者用 easy_install 安装它:

```
$ easy_install Flask-Exceptional
```

## 快速入门

在安装 Flask-Exceptional 后，所有你必须要做的就是创建一个 Flask 应用程序，配置 Exceptional API 密钥，接着创建 `Exceptional` 对象。正是这样的简单:

```
from flask import Flask
from flask.ext.exceptional import Exceptional

app = Flask(__name__)
app.config["EXCEPTIONAL_API_KEY"] = "exceptional_forty_character_unique_key"
exceptional = Exceptional(app)
```

你的应用程序是被配置成基于云的错误监控！你可以通过调用 `Exceptional.test()` 方法来验证配置是否正常工作:

```
Exceptional.test(app.config)
```

请翻阅接下来的章节获取更多关于 Flask-Exceptional 可用的配置项的细节。

## 配置

Flask-Exceptional 中存在如下的配置项:

| | |
|---|---|
| `EXCEPTIONAL_API_KEY` | 你的应用程序的Exceptional API 密钥。 登录到 Exceptional，选择你的应用程序， 点击 *APP SETTINGS* 链接。显示的 API 密钥就是这里要用到的。试图不提供 API 密钥而创建扩展将导致 登录警告，但应用程序将继续正常运行。 |
| `EXCEPTIONAL_DEBUG_URL` | 如果你的应用程序以调式模式运行的话，Exceptional 将不会捕获错误。配置这个值 是为了在调试模式中捕获错误数据。 比如，你可能使用一个 RequestBin 网址 调试你的应用程序。JSON 错误数据会被以压缩形式 POSTed 到这个网址， 而 Exceptional 需要解压这些数据。 |
| `EXCEPTIONAL_HTTP_CODES` | 用 Exceptional 追踪的 HTTP 错误码列表。默认为标准的 HTTP 4xx 错误码。 |
| `EXCEPTIONAL_PARAMETER_FILTER` | 列表值，用来过滤发给 Exceptional 的参数数据。 参数数据包括 `request.form` 和 `request.files` 中的所有。例如，为了过滤密码你可以使用： `['password', 'password_confirm']` |
| `EXCEPTIONAL_ENVIRONMENT_FILTER` | 列表值，用来过滤发给 Exceptional 的环境数据。 环境数据包含 Flask 应用程序配置以及目前 OS 环境。OS 环境值前缀是 `'os.'` 。例如，为了过滤 SQLAlchemy 数据库 URL以及 所有的 OS 环境值，使用： `['SQLALCHEMY_DATABASE_URI', 'os.*']` 默认值是 `['SECRET_KEY']` 。 |
| `EXCEPTIONAL_SESSION_FILTER` | 列表值，用来过滤发给 Exceptional 的会话数据。 |
| `EXCEPTIONAL_HEADER_FILTER` | 列表值，用来过滤发给 Exceptional 的 HTTP 头数据。 |
| `EXCEPTIONAL_COOKIE_FILTER` | 名称的列表，用来过滤发给 Exceptional 的 HTTP Cookie 头数据。 |

Note

所有配置中的过滤列表接受字符串以及正则表达式。

# API

`class flask.ext.exceptional.Exceptional(app=None)`

Extension for tracking application errors with Exceptional. Errors are not tracked if DEBUG is True. The application will log a warning if no `EXCEPTIONAL_API_KEY` has been configured.

Parameters:

**app** – Default None. The Flask application to track errors for. If the app is not provided on creation, then it can be provided later via `init_app()` .

`static context(data=None, **kwargs)`

Add extra context data to the current tracked exception. The context data is only valid for the current request. Multiple calls to this method will update any existing context with new data.

Parameters:

- **data** – Default `None` . A dictionary of context data.
- ****kwargs** – A series of keyword arguments to use as context data.

`init_app(app)`

Initialize this Exceptional extension.

Parameters:

**app** – The Flask application to track errors for.

`static publish(config, traceback)`

Publish the given traceback directly to Exceptional. This method is useful for tracking errors that occur outside the context of a Flask request. For example, this may be called from an asynchronous queue.

Parameters:

- **config** – A Flask application configuration object. Accepts either `flask.Config` or the object types allowed by `flask.Config.from_object()` .
- **traceback** – A `werkzeug.debug.tbtools.Traceback` instance to publish.

`static test(config)`

Test the given Flask configuration. If configured correctly, an error will be tracked by Exceptional for your app. Unlike the initialized extension, this test will post data to Exceptional, regardless of the configured `DEBUG` setting.

Parameters:

**config** – The Flask application configuration object to test. Accepts either `flask.Config` or the object types allowed by `flask.Config.from_object()` .

# Changelog

## Version 0.5.4

- Updated JSON implementation import to work with Flask 0.10.

## Version 0.5.3

- Fixed `Exceptional.publish()` to no longer dereference a request context.

## Version 0.5.2

- Unwind broken _app_ctx_stack usage.

## Version 0.5.1

- Handle malformed HTTP response status-line from Exceptional.

## Version 0.5

- Updated with Flask 0.8 extension structure recommendations and 0.9 _app_ctx_stack.
- Added `{'application_environment': 'loaded_libraries': {...}}` API data.

## Version 0.4.9

- Added the `Exceptional.context()` method to support Exceptional's extra context data API.
- Updated to reference the new exceptional.io domain.

## Version 0.4.8

- Updated to publish UTF-8 encoded data to Exceptional.
- Added support for `request.json` data.

## Version 0.4.7

- Added the `Exceptional.publish()` method to support Exceptional tracking outside the context of a request.

## Version 0.4.6

- Corrected `occurred_at` timestamp to be formatted as Zulu.
- Fixed JSON serialization issue by coercing all environment variables to strings.

## Version 0.4.5

- Updated to log a warning on repeated extension initialization attempts.

## Version 0.4.4

- Fixed to workaround Python 2.5 issue where `urlopen()` raises a `HTTPError` even though the HTTP response code indicates success.

## Version 0.4.3

- Changed so that `app.extensions['exceptional']` targets the `Exceptional` extension instance.

## Version 0.4.2

- Updated to support Python 2.5.

## Version 0.4.1

- Updated to support Flask 0.7 blueprints.

## Version 0.4

- Updated to support Python 2.6.
- Added `EXCEPTIONAL_DEBUG_URL` testing environment variable override.

## Version 0.3

- Updated to handle unreachable Exceptional service API.

## Version 0.2

- Added `Exceptional.test()` method.

## Version 0.1

- Initial public release.

# Flask-Login

Flask-Login 为 Flask 提供了用户会话管理。它处理了日常的登入，登出并且长时间记住用户的会话。

它会:

- 在会话中存储当前活跃的用户 ID，让你能够自由地登入和登出。
- 让你限制登入(或者登出)用户可以访问的视图。
- 处理让人棘手的 “记住我” 功能。
- 帮助你保护用户会话免遭 cookie 被盗的牵连。
- 可以与以后可能使用的 Flask-Principal 或其它认证扩展集成。

但是，它不会:

- 限制你使用特定的数据库或其它存储方法。如何加载用户完全由你决定。
- 限制你使用用户名和密码，OpenIDs，或者其它的认证方法。
- 处理超越 “登入或者登出” 之外的权限。
- 处理用户注册或者账号恢复。

# 配置你的应用

对一个使用 Flask-Login 的应用最重要的一部分就是 `LoginManager` 类。你应该在你的代码的某处为应用创建一个，像这样:

```
login_manager = LoginManager()
```

登录管理(login manager)包含了让你的应用和 Flask-Login 协同工作的代码，比如怎样从一个 ID 加载用户，当用户需要登录的时候跳转到哪里等等。

一旦实际的应用对象创建后，你能够这样配置它来实现登录:

```
login_manager.init_app(app)
```

# 它是如何工作

你必须提供一个 `user_loader` 回调。这个回调用于从会话中存储的用户 ID 重新加载用户对象。它应该接受一个用户的 `unicode` ID 作为参数，并且返回相应的用户对象。比如:

```
@login_manager.user_loader
def load_user(userid):
    return User.get(userid)
```

如果 ID 无效的话，它应该返回 `None` (而不是抛出异常)。(在这种情况下，ID 会被手动从会话中移除且处理会继续)

# 你的用户类

你用来表示用户的类需要实现这些属性和方法:

`is_authenticated`

当用户通过验证时，也即提供有效证明时返回 `True` 。（只有通过验证的用户会满足 `login_required` 的条件。）

`is_active`

如果这是一个活动用户且通过验证，账户也已激活，未被停用，也不符合任何你 的应用拒绝一个账号的条件，返回 `True` 。不活动的账号可能不会登入（当然， 是在没被强制的情况下）。

`is_anonymous`

如果是一个匿名用户，返回 `True` 。（真实用户应返回 `False` 。）

`get_id()`

返回一个能唯一识别用户的，并能用于从 `user_loader` 回调中加载用户的 `unicode` 。注意着 必须 是一个 `unicode` —— 如果 ID 原本是 一个 `int` 或其它类型，你需要把它转换为 `unicode` 。

要简便地实现用户类，你可以从 `UserMixin` 继承，它提供了对所有这些方法的默认 实现。（虽然这不是必须的。）

# Login 示例

一旦用户通过验证，你可以使用 `login_user` 函数让用户登录。例如:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    # Here we use a class of some kind to represent and validate our
    # client-side form data. For example, WTForms is a library that will
    # handle this for us, and we use a custom LoginForm to validate.
    form = LoginForm()
    if form.validate_on_submit():
        # Login and validate the user.
        # user should be an instance of your `User` class
        login_user(user)

        flask.flash('Logged in successfully.')

        next = flask.request.args.get('next')
        # next_is_valid should check if the user has valid
        # permission to access the `next` url
        if not next_is_valid(next):
            return flask.abort(400)

        return flask.redirect(next or flask.url_for('index'))
    return flask.render_template('login.html', form=form)
```

警告: 你必须验证 `next` 参数的值。如果不验证的话，你的应用将会受到重定向的攻击。

就这么简单。你可用使用 `current_user` 代理来访问登录的用户，在每一个模板中都可以使用 `current_user` :

```
{% if current_user.is_authenticated() %}
  Hi {{ current_user.name }}!
{% endif %}
```

需要用户登入 的视图可以用 `login_required` 装饰器来装饰:

```
@app.route("/settings")
@login_required
def settings():
    pass
```

当用户要登出时:

```
@app.route("/logout")
@login_required
def logout():
    logout_user()
    return redirect(somewhere)
```

他们会被登出，且他们会话产生的任何 cookie 都会被清理干净。

# 定制登入过程

默认情况下，当未登录的用户尝试访问一个 `login_required` 装饰的视图，Flask-Login 会闪现一条消息并且重定向到登录视图。(如果未设置登录视图，它将会以 401 错误退出。)

登录视图的名称可以设置成 `LoginManager.login_view`。例如:

```
login_manager.login_view = "users.login"
```

默认的闪现消息是 `Please log in to access this page.`。要自定义该信息，请设置 `LoginManager.login_message`：

```
login_manager.login_message = u"Bonvolu ensaluti por uzi tio pa臙o."
```

要自定义消息分类的话，请设置 `LoginManager.login_message_category`：

```
login_manager.login_message_category = "info"
```

当重定向到登入视图，它的请求字符串中会有一个 `next` 变量，其值为用户之前访问的页面。

如果你想要进一步自定义登入过程，请使用 `LoginManager.unauthorized_handler` 装饰函数:

```
@login_manager.unauthorized_handler
def unauthorized():
    # do stuff
    return a_response
```

# 使用授权头(Authorization header)登录

Caution

该方法将会被弃用，使用下一节的 `request_loader` 来代替。

有些时候你要支持使用 `Authorization` 头的基本认证登录，比如 API 请求。为了支持通过头登录你需要提供一个 `header_loader` 回调。这个回调和 `user_loader` 回调作用一样，只是它接受一个 HTTP 头(Authorization)而不是用户 id。例如:

```
@login_manager.header_loader
def load_user_from_header(header_val):
    header_val = header_val.replace('Basic ', '', 1)
    try:
        header_val = base64.b64decode(header_val)
    except TypeError:
        pass
    return User.query.filter_by(api_key=header_val).first()
```

默认情况下 `Authorization` 的值传给 `header_loader` 回调。你可以使用 `AUTH_HEADER_NAME` 配置来修改使用的 HTTP 头(可以不使用 `Authorization`，使用 `Token`)。

## 使用 Request Loader 定制登录

有时你想要不使用 cookies 情况下登录用户，比如使用 HTTP 头或者一个作为查询参数的 api 密钥。这种情况下，你应该使用 `request_loader` 回调。这个回调和 `user_loader` 回调作用一样，但是 `user_loader` 回调只接受 Flask 请求而不是一个 user_id。

例如，为了同时支持一个 url 参数和使用 `Authorization` 头的基本用户认证的登录:

```
@login_manager.request_loader
def load_user_from_request(request):

    # first, try to login using the api_key url arg
    api_key = request.args.get('api_key')
    if api_key:
        user = User.query.filter_by(api_key=api_key).first()
        if user:
            return user

    # next, try to login using Basic Auth
    api_key = request.headers.get('Authorization')
    if api_key:
        api_key = api_key.replace('Basic ', '', 1)
        try:
            api_key = base64.b64decode(api_key)
        except TypeError:
            pass
        user = User.query.filter_by(api_key=api_key).first()
        if user:
            return user

    # finally, return None if both methods did not login the user
    return None
```

## 匿名用户

默认情况下，当一个用户没有真正地登录，`current_user` 被设置成一个 `AnonymousUserMixin` 对象。它由如下的属性和方法:

- `is_active` 和 `is_authenticated` 的值为 `False`
- `is_anonymous` 的值为 `True`
- `get_id()` 返回 `None`

如果需要为匿名用户定制一些需求(比如，需要一个权限域)，你可以向 `LoginManager` 提供一个创建匿名用户的回调（类或工厂函数）：

```
login_manager.anonymous_user = MyAnonymousUser
```

# 记住我

“记住我”的功能很难实现。但是，Flask-Login 几乎透明地实现它 - 只要把 `remember=True` 传递给 `login_user` 。一个 cookie 将会存储在用户计算机中，如果用户会话中没有用户 ID 的话，Flask-Login 会自动地从 cookie 中恢复用户 ID。cookie 是防篡改的，因此如果用户篡改过它(比如，使用其它的一些东西来代替用户的 ID)，它就会被拒绝，就像不存在。

该层功能是被自动实现的。但你能（且应该，如果你的应用处理任何敏感的数据）提供 额外基础工作来增强你记住的 cookie 的安全性。

## 可选令牌

使用用户 ID 作为记住的令牌值不一定是安全的。更安全的方法是使用用户名和密码联合的 hash 值，或类似的东西。要添加一个额外的令牌，向你的用户对象添加一个方法：

`get_auth_token()`

返回用户的认证令牌（返回为 `unicode` ）。这个认证令牌应能唯一识别用户，且不易通过用户的公开信息，如 UID 和名称来猜测出——同样也不应暴露这些信息。

相应地，你应该在 `LoginManager` 上设置一个 `token_loader` 函数， 它接受令牌（存储在 cookie 中）作为参数并返回合适的 User 对象。

`make_secure_token` 函数用于便利创建认证令牌。它会连接所有的参数，然后用应用的密钥来 HMAC 它确保最大的密码学安全。（如果你永久地在数据库中存储用户令牌，那么你会希望向令牌中添加随机数据来阻碍猜测。）

如果你的应用使用密码来验证用户，在认证令牌中包含密码（或你应使用的加盐值的密码 hash ）能确保若用户更改密码，他们的旧认证令牌会失效。

## ”新鲜的“登录(Fresh Logins)

当用户登入，他们的会话被标记成“新鲜的”，就是说在这个会话只中用户实际上登录过。当会话销毁用户使用“记住我”的 cookie 重新登入，会话被标记成“非新鲜的”。 `login_required` 并不在意它们之间的不同，这适用于大部分页面。然而，更改某人 的个人信息这样的敏感操作应需要一个“新鲜的”的登入。（像修改密码这样的操作总是需要 密码，无论是否重登入。）

`fresh_login_required` ，除了验证用户登录，也将确保他们的登录是“新鲜的”。如果不是“新鲜的”，它会把用户送到可以重输入验证条件的页面。你可以定制 `fresh_login_required` 就像定制 `login_required` 那样，通过设置 `LoginManager.refresh_view` ， `needs_refresh_message` ，和 `needs_refresh_message_category` ：

```
login_manager.refresh_view = "accounts.reauthenticate"
login_manager.needs_refresh_message = (
    u"To protect your account, please reauthenticate to access this page."
)
login_manager.needs_refresh_message_category = "info"
```

或者提供自己的回调来处理“非新鲜的”刷新:

```
@login_manager.needs_refresh_handler
def refresh():
    # do stuff
    return a_response
```

调用 `confirm_login` 函数可以重新标记会话为”新鲜“。

## Cookie 设置

cookie 的细节可以在应用设置中定义。

| | |
|---|---|
| `REMEMBER_COOKIE_NAME` | 存储“记住我”信息的 cookie 名。 默认值： remember_token |
| `REMEMBER_COOKIE_DURATION` | cookie 过期时间，为一个 `datetime.timedelta` 对象。 默认值： 365 天 (1 非闰阳历年) |
| `REMEMBER_COOKIE_DOMAIN` | 如果“记住我” cookie 应跨域，在此处设置域名值 （即 .example.com 会允许 example 下所有子域 名）。 默认值： None |
| `REMEMBER_COOKIE_PATH` | 限制”记住我“ cookie 存储到某一路径下。 默认值： `/` |
| `REMEMBER_COOKIE_SECURE` | 限制 “Remember Me” cookie 在某些安全通道下有用 （典型地 HTTPS）。默认值： `None` |
| `REMEMBER_COOKIE_HTTPONLY` | 保护 “Remember Me” cookie 不能通过客户端脚本访问。 默认值： `False` |

# 会话保护

当上述特性保护“记住我”令牌免遭 cookie 窃取时，会话 cookie 仍然是脆弱的。 Flask-Login 包含了会话保护来帮助阻止用户会话被盗用。

你可以在 `LoginManager` 上和应用配置中配置会话保护。如果它被启用，它可以在 `basic` 或 `strong` 两种模式中运行。要在 `LoginManager` 上设置它，设置 `session_protection` 属性为 `"basic"` 或 `"strong"` ：

```
login_manager.session_protection = "strong"
```

或者，禁用它:

```
login_manager.session_protection = None
```

默认，它被激活为 `"basic"` 模式。它可以在应用配置中设定 `SESSION_PROTECTION` 为 `None` 、 `"basic"` 或 `"strong"` 来禁用。

当启用了会话保护，每个请求，它生成一个用户电脑的标识（基本上是 IP 地址和 User Agent 的 MD5 hash 值）。如果会话不包含相关的标识，则存储生成的。如果存在标识，则匹配生成的，之后请求可用。

在 `basic` 模式下或会话是永久的，如果该标识未匹配，会话会简单地被标记为非活 跃的，且任何需要活跃登入的东西会强制用户重新验证。（当然，你必须已经使用了活跃登入机制才能奏效。）

在 `strong` 模式下的非永久会话，如果该标识未匹配，整个会话（记住的令牌如果存在，则同样）被删除。

# 本地化

默认情况下，当用户需要登录， `LoginManager` 使用 `flash` 来显示信息。这些信息都是英文的。如果你需要本地化，设置 `LoginManager` 的 `localize_callback` 属性为一个函数，该函数在消息被发送到 `flash` 的时候被调用，比如， `gettext` 。

# API 文档

这部分文档是从 Flask-Login 源码中自动生成的。

## 配置登录

`class flask.ext.login.LoginManager(app=None, add_context_processor=True)`

This object is used to hold the settings used for logging in. Instances of `LoginManager` are *not* bound to specific apps, so you can create one in the main body of your code and then bind it to your app in a factory function.

`setup_app(app, add_context_processor=True)`

This method has been deprecated. Please use `LoginManager.init_app()` instead.

`unauthorized()`

This is called when the user is required to log in. If you register a callback with `LoginManager.unauthorized_handler()`, then it will be called. Otherwise, it will take the following actions:

- Flash `LoginManager.login_message` to the user.
- If the app is using blueprints find the login view for the current blueprint using `blueprint_login_views`. If the app is not using blueprints or the login view for the current blueprint is not specified use the value of `login_view`. Redirect the user to the login view. (The page they were attempting to access will be passed in the `next` query string variable, so you can redirect there if present instead of the homepage.)

If `LoginManager.login_view` is not defined, then it will simply raise a HTTP 401 (Unauthorized) error instead.

This should be returned from a view or before/after_request function, otherwise the redirect will have no effect.

`needs_refresh()`

This is called when the user is logged in, but they need to be reauthenticated because their session is stale. If you register a callback with `needs_refresh_handler`, then it will be called. Otherwise, it will take the following actions:

- Flash `LoginManager.needs_refresh_message` to the user.
- Redirect the user to `LoginManager.refresh_view`. (The page they were attempting to access will be passed in the `next` query string variable, so you can redirect there if present instead of the homepage.)

If `LoginManager.refresh_view` is not defined, then it will simply raise a HTTP 401 (Unauthorized) error instead.

This should be returned from a view or before/after_request function, otherwise the redirect will have no effect.

General Configuration

`user_loader(callback)`

This sets the callback for reloading a user from the session. The function you set should take a user ID (a `unicode` ) and return a user object, or `None` if the user does not exist.

Parameters: **callback** (*callable*) – The callback for retrieving a user object.

`header_loader(callback)`

This sets the callback for loading a user from a header value. The function you set should take an authentication token and return a user object, or `None` if the user does not exist.

Parameters: **callback** (*callable*) – The callback for retrieving a user object.

`token_loader(callback)`

This sets the callback for loading a user from an authentication token. The function you set should take an authentication token (a `unicode` , as returned by a user's `get_auth_token` method) and return a user object, or `None` if the user does not exist.

Parameters: **callback** (*callable*) – The callback for retrieving a user object.

`anonymous_user`

A class or factory function that produces an anonymous user, which is used when no one is logged in.

`unauthorized` Configuration

`login_view`

The name of the view to redirect to when the user needs to log in. (This can be an absolute URL as well, if your authentication machinery is external to your application.)

`login_message`

The message to flash when a user is redirected to the login page.

`unauthorized_handler(callback)`

This will set the callback for the `unauthorized` method, which among other things is used by `login_required` . It takes no arguments, and should return a response to be sent to the user instead of their normal view.

Parameters: **callback** (*callable*) – The callback for unauthorized users.

`needs_refresh` Configuration

`refresh_view`

The name of the view to redirect to when the user needs to reauthenticate.

`needs_refresh_message`

The message to flash when a user is redirected to the reauthentication page.

`needs_refresh_handler(callback)`

This will set the callback for the `needs_refresh` method, which among other things is used by `fresh_login_required`. It takes no arguments, and should return a response to be sent to the user instead of their normal view.

Parameters: **callback** (*callable*) – The callback for unauthorized users.

## 登录机制

`flask.ext.login.current_user`

A proxy for the current user.

`flask.ext.login.login_fresh()`

This returns `True` if the current login is fresh.

`flask.ext.login.login_user(user, remember=False, force=False, fresh=True)`

Logs a user in. You should pass the actual user object to this. If the user's `is_active` property is `False`, they will not be logged in unless `force` is `True`.

This will return `True` if the log in attempt succeeds, and `False` if it fails (i.e. because the user is inactive).

Parameters:

- **user** (*object*) – The user object to log in.
- **remember** (*bool*) – Whether to remember the user after their session expires. Defaults to `False`.
- **force** (*bool*) – If the user is inactive, setting this to `True` will log them in regardless. Defaults to `False`.
- **fresh** – setting this to `False` will log in the user with a session

marked as not “fresh”. Defaults to `True`. :type fresh: bool

`flask.ext.login.logout_user()`

Logs a user out. (You do not need to pass the actual user.) This will also clean up the remember me cookie if it exists.

`lask.ext.login.confirm_login()`

This sets the current session as fresh. Sessions become stale when they are reloaded from a cookie.

## 保护视图

`flask.ext.login.login_required(func)`

If you decorate a view with this, it will ensure that the current user is logged in and authenticated before calling the actual view. (If they are not, it calls the `LoginManager.unauthorized` callback.) For example:

```
@app.route('/post')
@login_required
def post():
    pass
```

If there are only certain times you need to require that your user is logged in, you can do so with:

```
if not current_user.is_authenticated:
    return current_app.login_manager.unauthorized()
```

...which is essentially the code that this function adds to your views.

It can be convenient to globally turn off authentication when unit testing. To enable this, if the application configuration variable `LOGIN_DISABLED` is set to `True` , this decorator will be ignored.

Parameters: **func** (*function*) – The view function to decorate.

`flask.ext.login.fresh_login_required(func)`

If you decorate a view with this, it will ensure that the current user's login is fresh - i.e. there session was not restored from a 'remember me' cookie. Sensitive operations, like changing a password or e-mail, should be protected with this, to impede the efforts of cookie thieves.

If the user is not authenticated, `LoginManager.unauthorized()` is called as normal. If they are authenticated, but their session is not fresh, it will call `LoginManager.needs_refresh()` instead. (In that case, you will need to provide a `LoginManager.refresh_view` .)

Behaves identically to the `login_required()` decorator with respect to configutation variables.

Parameters: **func** (*function*) – The view function to decorate.

## 用户对象助手

`class flask.ext.login.UserMixin`

This provides default implementations for the methods that Flask-Login expects user objects to have.

## 工具

`flask.ext.login.login_url(login_view, next_url=None, next_field='next')`

Creates a URL for redirecting to a login page. If only `login_view` is provided, this will just return the URL for it. If `next_url` is provided, however, this will append a `next=URL` parameter to the query string so that the login view can redirect back to that URL.

Parameters:

- **login_view** (*str*) – The name of the login view. (Alternately, the actual URL to the login view.)
- **next_url** (*str*) – The URL to give the login view for redirection.
- **next_field** (*str*) – What field to store the next URL in. (It defaults to `next`.)

`flask.ext.login.make_secure_token(*args, **options)`

This will create a secure token that you can use as an authentication token for your users. It uses heavy-duty HMAC encryption to prevent people from guessing the information. (To make it even more effective, if you will never need to regenerate the token, you can pass some random data as one of the arguments.)

Parameters:

- ***args** – The data to include in the token.
- ****options** (*kwargs*) – To manually specify a secret key, pass `key=THE_KEY`. Otherwise, the `current_app` secret key will be used.

## 信号

如何在你的代码中使用这些信号请参阅 Flask documentation on signals。

`flask.ext.login.user_logged_in`

当一个用户登入的时候发出。除应用（信号的发送者）之外，它还传递正登入的用户 `user` 。

`flask.ext.login.user_logged_out`

当一个用户登出的时候发出。除应用（信号的发送者）之外，它还传递正登出的用户 `user` 。

`flask.ext.login.user_login_confirmed`

当用户的登入被证实，把它标记为活跃的。（它不用于常规登入的调用。） 它不接受应用以外的任何其它参数。

`flask.ext.login.user_unauthorized`

当 `LoginManager` 上的 `unauthorized` 方法被调用时发出。它不接受应用以外的任何其它参数。

`flask.ext.login.user_needs_refresh`

当 `LoginManager` 上的 `needs_refresh` 方法被调用时发出。它不接受应用以外的任何其它参数。

`flask.ext.login.session_protected`

当会话保护起作用时，且会话被标记为非活跃或删除时发出。它不接受应用以外的任何其它参数。

# flask-mail

在 Web 应用中的一个最基本的功能就是能够给用户发送邮件。

**Flask-Mail** 扩展提供了一个简单的接口，可以在 Flask 应用中设置 SMTP 使得可以在视图以及脚本中发送邮件信息。

## 链接

- 原版文档
- 源代码
- 更新历史

## 安装 Flask-Mail

使用 **pip** 或者 **easy_install** 安装 Flask-Mail:

```
pip install Flask-Mail
```

或者从版本控制系统（github）中下载最新的版本:

```
git clone https://github.com/mattupstate/flask-mail.git
cd flask-mail
python setup.py install
```

如果你正在使用 **virtualenv**，假设你会安装 flask-mail 在运行你的 Flask 应用程序的同一个 virtualenv 上。

## 配置 Flask-Mail

**Flask-Mail** 使用标准的 Flask 配置 API 进行配置。下面这些是可用的配置型(每一个将会在文档中进行解释):

- **MAIL_SERVER** : 默认为 **'localhost'**
- **MAIL_PORT** : 默认为 **25**
- **MAIL_USE_TLS** : 默认为 **False**
- **MAIL_USE_SSL** : 默认为 **False**
- **MAIL_DEBUG** : 默认为 **app.debug**
- **MAIL_USERNAME** : 默认为 **None**

- **MAIL_PASSWORD** : 默认为 **None**
- **MAIL_DEFAULT_SENDER** : 默认为 **None**
- **MAIL_MAX_EMAILS** : 默认为 **None**
- **MAIL_SUPPRESS_SEND** : 默认为 **app.testing**
- **MAIL_ASCII_ATTACHMENTS** : 默认为 **False**

另外，**Flask-Mail** 使用标准的 Flask 的 `TESTING` 配置项用于单元测试(下面会具体介绍)。

邮件是通过一个 `Mail` 实例进行管理:

```
from flask import Flask
from flask_mail import Mail

app = Flask(__name__)
mail = Mail(app)
```

在这个例子中所有的邮件将会使用传入到 `Mail` 实例中的应用程序的配置项进行发送。

或者你也可以在应用程序配置的时候设置你的 `Mail` 实例，通过使用 **init_app** 方法:

```
mail = Mail()

app = Flask(__name__)
mail.init_app(app)
```

在这个例子中邮件将会使用 Flask 的 `current_app` 中的配置项进行发送。如果你有多个具有不用配置项的多个应用运行在同一程序的时候，这种设置方式是十分有用的，

# 发送邮件

为了能够发送邮件，首先需要创建一个 `Message` 实例:

```
from flask_mail import Message

@app.route("/")
def index():

    msg = Message("Hello",
                  sender="from@example.com",
                  recipients=["to@example.com"])
```

你能够设置一个或者多个收件人:

```
msg.recipients = ["you@example.com"]
msg.add_recipient("somebodyelse@example.com")
```

如果你设置了 `MAIL_DEFAULT_SENDER` ，就不必再次填写发件人，默认情况下将会使用配置项的发件人:

```
msg = Message("Hello",
              recipients=["to@example.com"])
```

如果 `sender` 是一个二元组，它将会被分成姓名和邮件地址:

```
msg = Message("Hello",
              sender=("Me", "me@example.com"))

assert msg.sender == "Me <me@example.com>"
```

邮件内容可以包含主体以及/或者 HTML:

```
msg.body = "testing"
msg.html = "<b>testing</b>"
```

最后，发送邮件的时候请使用 Flask 应用设置的 `Mail` 实例:

```
mail.send(msg)
```

# 大量邮件

通常在一个 Web 应用中每一个请求会同时发送一封或者两封邮件。在某些特定的场景下，有可能会发送数十或者数百封邮件，不过这种发送工作会给交离线任务或者脚本执行。

在这种情况下发送邮件的代码会有些不同:

```
with mail.connect() as conn:
    for user in users:
        message = '...'
        subject = "hello, %s" % user.name
        msg = Message(recipients=[user.email],
                      body=message,
                      subject=subject)

        conn.send(msg)
```

与电子邮件服务器的连接会一直保持活动状态直到所有的邮件都已经发送完成后才会关闭（断开）。

有些邮件服务器会限制一次连接中的发送邮件的上限。你可以设置重连前的发送邮件的最大数，通过配置 **MAIL_MAX_EMAILS** 。

# 附件

在邮件中添加附件同样非常简单:

```
with app.open_resource("image.png") as fp:
    msg.attach("image.png", "image/png", fp.read())
```

具体细节请参看 API 。

如果 `MAIL_ASCII_ATTACHMENTS` 设置成 **True** 的话，文件名将会转换成 ASCII 的。 当文件名是以 UTF-8 编码的时候，使用邮件转发的时候会修改邮件内容并且混淆 Content-Disposition 描述，这个时候 `MAIL_ASCII_ATTACHMENTS` 配置项是十分有用的。转换成 ASCII 的基本方式就是对 non-ASCII 字符的去除。任何一个 unicode 字符能够被 NFKD 分解成一个或者多个 ASCII 字符。

# 单元测试以及禁止发送邮件

当在单元测试中，或者在一个开发环境中，能够禁止邮件发送是十分有用的。

如果设置项 `TESTING` 设置成 `True` ，emails 将会被禁止发送。调用 `send()` 发送邮件实际上不会有任何邮件被发送。

另外在测试环境之中的话，你可以设置 `MAIL_SUPPRESS_SEND` 为 **True**，这也会有相同的效果。

然而，当单元测试的时候追踪邮件是否发送成功也是十分有用的。

为了能够追踪发送邮件的“轨迹”，可以使用 `record_messages` 方法:

```
with mail.record_messages() as outbox:

    mail.send_message(subject='testing',
                      body='test',
                      recipients=emails)

    assert len(outbox) == 1
    assert outbox[0].subject == "testing"
```

**outbox** 是一个 发送 `Message` 实例的列表。

为了使得上述代码能够正常运行，必须安装 blinker 包。

需要注意的是以前的处理方式，即把 **outbox** 赋予给 `g` 对象已经过时。

# 头注入

为了防止 header injection，尝试着在邮件主题、发件人或者收件人中使用换行符将会抛出 `BadHeaderError` 异常。

# 信号量

New in version 0.4.

**Flask-Mail** 现在通过 `email_dispatched` 信号开始支持信号量。只要邮件被调度，信号就会发送(即使邮件没有真正的发送，例如，在测试环境中)。

订阅 `email_dispatched` 信号的函数使用 `Message` 实例作为第一参数，Flask app 实例作为一个可选的参数:

```
def log_message(message, app):
    app.logger.debug(message.subject)

email_dispatched.connect(log_message)
```

# API

`class flask_mail.Mail(app=None)`

Manages email messaging

Parameters: **app** – Flask instance

`connect()`

Opens a connection to the mail host.

`send(message)`

Sends a single message instance. If TESTING is True the message will not actually be sent.

Parameters: **message** – a Message instance.

`send_message(*args, **kwargs)`

Shortcut for send(msg).

Takes same arguments as Message constructor.

Versionadded: 0.3.5

`class flask_mail.Attachment(filename=None, content_type=None, data=None, disposition=None,`

Encapsulates file attachment information.

Versionadded:

0.3.5

Parameters:

- **filename** – filename of attachment
- **content_type** – file mimetype

- **data** – the raw file data
- **disposition** – content-disposition (if any)

```
class flask_mail.Connection(mail)
```

Handles connection to host.

```
send(message, envelope_from=None)
```

Verifies and sends message.

Parameters:

- **message** – Message instance.
- **envelope_from** – Email address to be used in MAIL FROM command.

```
send_message(*args, **kwargs)
```

Shortcut for send(msg).

Takes same arguments as Message constructor.

Versionadded: 0.3.5

```
class flask_mail.Message(subject='', recipients=None, body=None, html=None, sender=None, c
```

Encapsulates an email message.

Parameters:

- **subject** – email subject header
- **recipients** – list of email addresses
- **body** – plain text message
- **html** – HTML message
- **sender** – email sender address, or **MAIL_DEFAULT_SENDER** by default
- **cc** – CC list
- **bcc** – BCC list
- **attachments** – list of Attachment instances
- **reply_to** – reply-to address
- **date** – send date
- **charset** – message character set
- **extra_headers** – A dictionary of additional headers for the message
- **mail_options** – A list of ESMTP options to be used in MAIL FROM command
- **rcpt_options** – A list of ESMTP options to be used in RCPT commands

```
add_recipient(recipient)
```

Adds another recipient to the message.

Parameters: **recipient** – email address of recipient.

`attach(filename=None, content_type=None, data=None, disposition=None, headers=None)`

Adds an attachment to the message.

Parameters:

- **filename** – filename of attachment
- **content_type** – file mimetype
- **data** – the raw file data
- **disposition** – content-disposition (if any)

# Flask-PyMongo

MongoDB 是一个开源的数据库，它存储着灵活的类-JSON 的“文档”。与关系数据库中的数据行相反，它能够存储任何的数字，名称，或者复杂的层级结构。Python 开发者可以考虑把 MongoDB 作为一个持久化，可搜索的 Python 字典的“仓库”(实际上，这是如何用 PyMongo 来表示 MongoDB 中的“文档”)。

Flask-PyMongo 架起来 Flask 和 PyMongo 之间的桥梁，因此你能够使用 Flask 正常的机制去配置和连接 MongoDB。

## 快速入门

首先，安装 Flask-PyMongo:

```
$ pip install Flask-PyMongo
```

Flask-PyMongo 的各种依赖，比如，最新版本的 Flask（0.8或者以上）以及 PyMongo（2.4或者以上），也会为你安装的。Flask-PyMongo 是兼容 Python 2.6, 2.7, 和 3.3 版本并且通过测试。

接着，在你的代码中添加一个 `PyMongo`:

```
from flask import Flask
from flask.ext.pymongo import PyMongo

app = Flask(__name__)
mongo = PyMongo(app)
```

`PyMongo` 连接运行在本机上且端口为 27017 的 MongoDB 服务器，并且假设默认的数据库名为 `app.name` (换而言之，你可以使用传入到 `Flask` 中的任何数据库名)。这个数据库能够作为 `db` 属性被导入。

你可以在视图中直接使用 `db` ：

```
@app.route('/')
def home_page():
    online_users = mongo.db.users.find({'online': True})
    return render_template('index.html',
        online_users=online_users)
```

## Helpers

Flask-PyMongo 提供一些通用任务的现成方法:

`Collection.find_one_or_404(*args, **kwargs)`

Find and return a single document, or raise a 404 Not Found exception if no document matches the query spec. See `find_one()` for details.

```
@app.route('/user/<username>')
def user_profile(username):
    user = mongo.db.users.find_one_or_404({'_id': username})
    return render_template('user.html',
        user=user)
```

`PyMongo.send_file(filename, base='fs', version=-1, cache_for=31536000)`

Return an instance of the `response_class` containing the named file, and implement conditional GET semantics (using `make_conditional()` ).

```
@app.route('/uploads/<path:filename>')
def get_upload(filename):
    return mongo.send_file(filename)
```

Parameters:

- **filename** (*str*) – the filename of the file to return
- **base** (*str*) – the base name of the GridFS collections to use
- **version** (*bool*) – if positive, return the Nth revision of the file identified by filename; if negative, return the Nth most recent revision. If no such version exists, return with HTTP status 404.
- **cache_for** (*int*) – number of seconds that browsers should be instructed to cache responses

`PyMongo.save_file(filename, fileobj, base='fs', content_type=None)`

Save the file-like object to GridFS using the given filename. Returns `None` .

```
@app.route('/uploads/<path:filename>', methods=['POST'])
def save_upload(filename):
    mongo.save_file(filename, request.files['file'])
    return redirect(url_for('get_upload', filename=filename))
```

Parameters:

- **filename** (*str*) – the filename of the file to return
- **fileobj** (*file*) – the file-like object to save
- **base** (*str*) – base the base name of the GridFS collections to use
- **content_type** (*str*) – the MIME content-type of the file. If `None` , the content-type is guessed from the filename using `guess_type()`

`class flask_pymongo.BSONObjectIdConverter(map)`

A simple converter for the RESTful URL routing system of Flask.

```
@app.route('/<ObjectId:task_id>')
def show_task(task_id):
    task = mongo.db.tasks.find_one_or_404(task_id)
    return render_template('task.html', task=task)
```

Valid object ID strings are converted into `ObjectId` objects; invalid strings result in a 404 error. The converter is automatically registered by the initialization of `PyMongo` with keyword `ObjectId`.

# Configuration

`PyMongo` 直接支持如下的配置项:

| | |
|---|---|
| `MONGO_URI` | 一个 MongoDB 网址 用于其他配置项。 |
| `MONGO_HOST` | 你的 MongoDB 服务器的主机名或者 IP 地址。 默认：”localh |
| `MONGO_PORT` | 你的 MongoDB 服务器的端口。默认：27017。 |
| `MONGO_AUTO_START_REQUEST` | 为了禁用 PyMongo 2.2 的 “auto start request” 行为， 设置成 `MongoClient` )。 默认： `True` 。 |
| `MONGO_MAX_POOL_SIZE` | (可选): PyMongo 连接池中保持空闲连接的最大数量。 默认： |
| `MONGO_SOCKET_TIMEOUT_MS` | (可选): (整型) 在超时前套接字允许一个发送或者接收的耗时( 值。 |
| `MONGO_CONNECT_TIMEOUT_MS` | (可选): (整型) 在超时前允许一个连接的耗时(毫秒)。 默认: Py |
| `MONGO_DBNAME` | 可用于作为 `db` 属性的数据库名。默认： `app.name` 。 |
| `MONGO_USERNAME` | 用于认证的用户名。默认： `None` 。 |
| `MONGO_PASSWORD` | 用于认证的密码。默认： `None` 。 |
| `MONGO_REPLICA_SET` | 设置成连接的备份集的名称； 这必须匹配到备份集的内部名， http://www.mongodb.org/display/DOCS/Replica+Set+Comma isMaster 命令)决定的。默认： `None` 。 |
| `MONGO_READ_PREFERENCE` | 决定如何读取路由到备份集的成员。 必须是定义在 `pymongo.read_preferences.ReadPreference` 中的一个常量 或者 |
| `MONGO_DOCUMENT_CLASS` | 告诉 pymongo 返回定制的对象而不是默认的字典， 比如 `bs` `dict` 。 |

当 `PyMongo` 或者 `init_app()` 仅仅只有一个参数调用的时候 (the Flask 实例)，会假设配置值的前缀是 `MONGO` ；能够用 `config_prefix` 来覆盖这个前缀。

这个技术能够用于连接多个数据库或者数据服务器:

```
app = Flask(__name__)

# connect to MongoDB with the defaults
mongo1 = PyMongo(app)

# connect to another MongoDB database on the same host
app.config['MONGO2_DBNAME'] = 'dbname_two'
mongo2 = PyMongo(app, config_prefix='MONGO2')

# connect to another MongoDB server altogether
app.config['MONGO3_HOST'] = 'another.host.example.com'
app.config['MONGO3_PORT'] = 27017
app.config['MONGO3_DBNAME'] = 'dbname_three'
mongo3 = PyMongo(app, config_prefix='MONGO3')
```

你应该需要注意一些自动配置的设置:

`tz_aware` :

Flask-PyMongo 一直使用通用时区的 `datetime` 对象。这是因为当建立连接的时候它设置 `tz_aware` 参数为 `True` 。从 MongoDB 返回的 `datetime` 对象一直是 UTC。

`safe` :

Flask-PyMongo 默认地设置成 “safe” 模式，这会导致 `save()` , `insert()` , `update()` , 和 `remove()` 在返回前一直等待着服务器的应答。你可以在调用的时候通过传入 `safe=False` 参数到任何一个受影响的方法中来覆盖它。

# Flask-RESTful

**Flask-RESTful** 是一个 Flask 扩展，它添加了快速构建 REST APIs 的支持。它当然也是一个能够跟你现有的ORM/库协同工作的轻量级的扩展。Flask-RESTful 鼓励以最小设置的最佳实践。如果你熟悉 Flask 的话，Flask-RESTful 应该很容易上手。

## 安装

使用 `pip` 安装 Flask-RESTful:

```
pip install flask-restful
```

开发的版本可以从 GitHub 上的页面 下载

```
git clone https://github.com/twilio/flask-restful.git
cd flask-restful
python setup.py develop
```

Flask-RESTful 有如下的依赖包(如果你使用 `pip` ，依赖包会自动地安装):

- Flask 版本 0.8 或者更高

Flask-RESTful 要求 Python 版本为 2.6, 2.7, 或者 3.3。

## 快速入门

是时候编写你第一个 REST API。本指南假设你对 Flask 有一定的认识，并且已经安装了 Flask 和 Flask-RESTful。如果还没有安装的话，可以依照 安装 章节的步骤安装。

### 一个最小的 API

一个最小的 Flask-RESTful API 像这样:

```
from flask import Flask
from flask.ext import restful

app = Flask(__name__)
api = restful.Api(app)

class HelloWorld(restful.Resource):
    def get(self):
        return {'hello': 'world'}

api.add_resource(HelloWorld, '/')

if __name__ == '__main__':
    app.run(debug=True)
```

把上述代码保存为 api.py 并且在你的 Python 解释器中运行它。需要注意地是我们已经启用了 Flask 调试 模式，这种模式提供了代码的重载以及更好的错误信息。调试模式绝不能在生产环境下使用。

```
$ python api.py
 * Running on http://127.0.0.1:5000/
```

现在打开一个新的命令行窗口使用 curl 测试你的 API:

```
$ curl http://127.0.0.1:5000/
{"hello": "world"}
```

# 资源丰富的路由(Resourceful Routing)

Flask-RESTful 提供的最主要的基础就是资源(resources)。资源(Resources)是构建在 Flask 可拔插视图 之上，只要在你的资源(resource)上定义方法就能够容易地访问多个 HTTP 方法。一个待办事项应用程序的基本的 CRUD 资源看起来像这样:

```
from flask import Flask, request
from flask.ext.restful import Resource, Api

app = Flask(__name__)
api = Api(app)

todos = {}

class TodoSimple(Resource):
    def get(self, todo_id):
        return {todo_id: todos[todo_id]}

    def put(self, todo_id):
        todos[todo_id] = request.form['data']
        return {todo_id: todos[todo_id]}

api.add_resource(TodoSimple, '/<string:todo_id>')

if __name__ == '__main__':
    app.run(debug=True)
```

你可以尝试这样:

```
$ curl http://localhost:5000/todo1 -d "data=Remember the milk" -X PUT
{"todo1": "Remember the milk"}
$ curl http://localhost:5000/todo1
{"todo1": "Remember the milk"}
$ curl http://localhost:5000/todo2 -d "data=Change my brakepads" -X PUT
{"todo2": "Change my brakepads"}
$ curl http://localhost:5000/todo2
{"todo2": "Change my brakepads"}
```

或者如果你安装了 requests 库的话，可以从 python shell 中运行:

```
>>> from requests import put, get
>>> put('http://localhost:5000/todo1', data={'data': 'Remember the milk'}).json()
{u'todo1': u'Remember the milk'}
>>> get('http://localhost:5000/todo1').json()
{u'todo1': u'Remember the milk'}
>>> put('http://localhost:5000/todo2', data={'data': 'Change my brakepads'}).json()
{u'todo2': u'Change my brakepads'}
>>> get('http://localhost:5000/todo2').json()
{u'todo2': u'Change my brakepads'}
```

Flask-RESTful 支持视图方法多种类型的返回值。同 Flask 一样，你可以返回任一迭代器，它将会被转换成一个包含原始 Flask 响应对象的响应。Flask-RESTful 也支持使用多个返回值来设置响应代码和响应头，如下所示:

```
class Todo1(Resource):
    def get(self):
        # Default to 200 OK
        return {'task': 'Hello world'}

class Todo2(Resource):
    def get(self):
        # Set the response code to 201
        return {'task': 'Hello world'}, 201

class Todo3(Resource):
    def get(self):
        # Set the response code to 201 and return custom headers
        return {'task': 'Hello world'}, 201, {'Etag': 'some-opaque-string'}
```

# 端点(Endpoints)

很多时候在一个 API 中，你的资源可以通过多个 URLs 访问。你可以把多个 URLs 传给 Api 对象的 `Api.add_resource()` 方法。每一个 URL 都能访问到你的 `Resource`

```
api.add_resource(HelloWorld,
    '/',
    '/hello')
```

你也可以为你的资源方法指定 endpoint 参数。

```
api.add_resource(Todo,
    '/todo/<int:todo_id>', endpoint='todo_ep')
```

# 参数解析

尽管 Flask 能够简单地访问请求数据(比如查询字符串或者 POST 表单编码的数据)，验证表单数据仍然很痛苦。Flask-RESTful 内置了支持验证请求数据，它使用了一个类似 argparse 的库。

```
from flask.ext.restful import reqparse

parser = reqparse.RequestParser()
parser.add_argument('rate', type=int, help='Rate to charge for this resource')
args = parser.parse_args()
```

需要注意地是与 argparse 模块不同，`reqparse.RequestParser.parse_args()` 返回一个 Python 字典而不是一个自定义的数据结构。

使用 `reqparse` 模块同样可以自由地提供聪明的错误信息。如果参数没有通过验证，Flask-RESTful 将会以一个 400 错误请求以及高亮的错误信息回应。

```
$ curl -d 'rate=foo' http://127.0.0.1:5000/
{'status': 400, 'message': 'foo cannot be converted to int'}
```

`inputs` 模块提供了许多的常见的转换函数，比如 `inputs.date()` 和 `inputs.url()`。

使用 `strict=True` 调用 `parse_args` 能够确保当请求包含你的解析器中未定义的参数的时候会抛出一个异常。

> args = parser.parse_args(strict=True)

# 数据格式化

默认情况下，在你的返回迭代中所有字段将会原样呈现。尽管当你刚刚处理 Python 数据结构的时候，觉得这是一个伟大的工作，但是当实际处理它们的时候，会觉得十分沮丧和枯燥。为了解决这个问题，Flask-RESTful 提供了 `fields` 模块和 `marshal_with()` 装饰器。类似 Django ORM 和 WTForm，你可以使用 fields 模块来在你的响应中格式化结构。

```
from collections import OrderedDict
from flask.ext.restful import fields, marshal_with

resource_fields = {
    'task':   fields.String,
    'uri':    fields.Url('todo_ep')
}

class TodoDao(object):
    def __init__(self, todo_id, task):
        self.todo_id = todo_id
        self.task = task

        # This field will not be sent in the response
        self.status = 'active'

class Todo(Resource):
    @marshal_with(resource_fields)
    def get(self, **kwargs):
        return TodoDao(todo_id='my_todo', task='Remember the milk')
```

上面的例子接受一个 python 对象并准备将其序列化。`marshal_with()` 装饰器将会应用到由 `resource_fields` 描述的转换。从对象中提取的唯一字段是 `task` 。`fields.Url` 域是一个特殊的域，它接受端点（endpoint）名称作为参数并且在响应中为该端点生成一个 URL。许多你需要的字段类型都已经包含在内。请参阅 `fields` 指南获取一个完整的列表。

# 完整的例子

在 api.py 中保存这个例子

```
from flask import Flask
from flask.ext.restful import reqparse, abort, Api, Resource

app = Flask(__name__)
api = Api(app)

TODOS = {
    'todo1': {'task': 'build an API'},
    'todo2': {'task': '?????'},
    'todo3': {'task': 'profit!'},
}

def abort_if_todo_doesnt_exist(todo_id):
    if todo_id not in TODOS:
        abort(404, message="Todo {} doesn't exist".format(todo_id))

parser = reqparse.RequestParser()
parser.add_argument('task', type=str)

# Todo
#   show a single todo item and lets you delete them
class Todo(Resource):
    def get(self, todo_id):
        abort_if_todo_doesnt_exist(todo_id)
        return TODOS[todo_id]

    def delete(self, todo_id):
        abort_if_todo_doesnt_exist(todo_id)
        del TODOS[todo_id]
        return '', 204

    def put(self, todo_id):
        args = parser.parse_args()
        task = {'task': args['task']}
        TODOS[todo_id] = task
        return task, 201

# TodoList
#   shows a list of all todos, and lets you POST to add new tasks
class TodoList(Resource):
    def get(self):
        return TODOS

    def post(self):
        args = parser.parse_args()
        todo_id = int(max(TODOS.keys()).lstrip('todo')) + 1
        todo_id = 'todo%i' % todo_id
        TODOS[todo_id] = {'task': args['task']}
        return TODOS[todo_id], 201

##
## Actually setup the Api resource routing here
##
api.add_resource(TodoList, '/todos')
api.add_resource(Todo, '/todos/<todo_id>')

if __name__ == '__main__':
    app.run(debug=True)
```

用法示例

```
$ python api.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

获取列表

```
$ curl http://localhost:5000/todos
{"todo1": {"task": "build an API"}, "todo3": {"task": "profit!"}, "todo2": {"task": "????
```

获取一个单独的任务

```
$ curl http://localhost:5000/todos/todo3
{"task": "profit!"}
```

删除一个任务

```
$ curl http://localhost:5000/todos/todo2 -X DELETE -v

> DELETE /todos/todo2 HTTP/1.1
> User-Agent: curl/7.19.7 (universal-apple-darwin10.0) libcurl/7.19.7 OpenSSL/0.9.8l zlib
> Host: localhost:5000
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 204 NO CONTENT
< Content-Type: application/json
< Content-Length: 0
< Server: Werkzeug/0.8.3 Python/2.7.2
< Date: Mon, 01 Oct 2012 22:10:32 GMT
```

增加一个新的任务

```
$ curl http://localhost:5000/todos -d "task=something new" -X POST -v

> POST /todos HTTP/1.1
> User-Agent: curl/7.19.7 (universal-apple-darwin10.0) libcurl/7.19.7 OpenSSL/0.9.8l zlib
> Host: localhost:5000
> Accept: */*
> Content-Length: 18
> Content-Type: application/x-www-form-urlencoded
>
* HTTP 1.0, assume close after body
< HTTP/1.0 201 CREATED
< Content-Type: application/json
< Content-Length: 25
< Server: Werkzeug/0.8.3 Python/2.7.2
< Date: Mon, 01 Oct 2012 22:12:58 GMT
<
* Closing connection #0
{"task": "something new"}
```

更新一个任务

```
$ curl http://localhost:5000/todos/todo3 -d "task=something different" -X PUT -v

> PUT /todos/todo3 HTTP/1.1
> Host: localhost:5000
> Accept: */*
> Content-Length: 20
> Content-Type: application/x-www-form-urlencoded
>
* HTTP 1.0, assume close after body
< HTTP/1.0 201 CREATED
< Content-Type: application/json
< Content-Length: 27
< Server: Werkzeug/0.8.3 Python/2.7.3
< Date: Mon, 01 Oct 2012 22:13:00 GMT
<
* Closing connection #0
{"task": "something different"}
```

# 请求解析

Flask-RESTful 的请求解析接口是模仿 `argparse` 接口。它设计成提供简单并且统一的访问 Flask 中 `flask.request` 对象里的任何变量的入口。

## 基本参数

这里是请求解析一个简单的例子。它寻找在 `flask.Request.values` 字典里的两个参数。一个类型为 `int` ，另一个的类型是 `str`

```
from flask.ext.restful import reqparse

parser = reqparse.RequestParser()
parser.add_argument('rate', type=int, help='Rate cannot be converted')
parser.add_argument('name', type=str)
args = parser.parse_args()
```

如果你指定了 help 参数的值，在解析的时候当类型错误被触发的时候，它将会被作为错误信息给呈现出来。如果你没有指定 help 信息的话，默认行为是返回类型错误本身的信息。

默认下，arguments 不是 必须的。另外，在请求中提供的参数不属于 RequestParser 的一部分的话将会被忽略。

另请注意：在请求解析中声明的参数如果没有在请求本身设置的话将默认为 `None` 。

## 必需的参数

要求一个值传递的参数，只需要添加 `required=True` 来调用 `add_argument()` 。

```
parser.add_argument('name', type=str, required=True,
help="Name cannot be blank!")
```

# 多个值&列表

如果你要接受一个键有多个值的话，你可以传入 `action='append'`

```
parser.add_argument('name', type=str, action='append')
```

这将让你做出这样的查询

```
curl http://api.example.com -d "Name=bob" -d "Name=sue" -d "Name=joe"
```

你的参数将会像这样

```
args = parser.parse_args()
args['name']    # ['bob', 'sue', 'joe']
```

# 其它目标（Destinations）

如果由于某种原因，你想要以不同的名称存储你的参数一旦它被解析的时候，你可以使用 `dest` kwarg。

```
parser.add_argument('name', type=str, dest='public_name')

args = parser.parse_args()
args['public_name']
```

# 参数位置

默认下， `RequestParser` 试着从 `flask.Request.values` ，以及 `flask.Request.json` 解析值。

在 `add_argument()` 中使用 `location` 参数可以指定解析参数的位置。 `flask.Request` 中任何变量都能被使用。例如：

```
# Look only in the POST body
parser.add_argument('name', type=int, location='form')

# Look only in the querystring
parser.add_argument('PageSize', type=int, location='args')

# From the request headers
parser.add_argument('User-Agent', type=str, location='headers')

# From http cookies
parser.add_argument('session_id', type=str, location='cookies')

# From file uploads
parser.add_argument('picture', type=werkzeug.datastructures.FileStorage, location='files'
```

# 多个位置

通过传入一个列表到 `location` 中可以指定 多个 参数位置:

```
parser.add_argument('text', location=['headers', 'values'])
```

列表中最后一个优先出现在结果集中。（例如：location=['headers', 'values']，解析后 'values' 的结果会在 'headers' 前面）

# 继承解析

往往你会为你编写的每个资源编写不同的解析器。这样做的问题就是如果解析器具有共同的参数。不是重写，你可以编写一个包含所有共享参数的父解析器接着使用 `copy()` 扩充它。你也可以使用 `replace_argument()` 覆盖父级的任何参数，或者使用 `remove_argument()` 完全删除参数。 例如：

```
from flask.ext.restful import RequestParser

parser = RequestParser()
parser.add_argument('foo', type=int)

parser_copy = parser.copy()
parser_copy.add_argument('bar', type=int)

# parser_copy has both 'foo' and 'bar'

parser_copy.replace_argument('foo', type=str, required=True, location='json')
# 'foo' is now a required str located in json, not an int as defined
#  by original parser

parser_copy.remove_argument('foo')
# parser_copy no longer has 'foo' argument
```

# 输出字段

Flask-RESTful 提供了一个简单的方式来控制在你的响应中实际呈现什么数据。使用 `fields` 模块，你可以使用在你的资源里的任意对象（ORM 模型、定制的类等等）并且 `fields` 让你格式化和过滤响应，因此您不必担心暴露内部数据结构。

当查询你的代码的时候，哪些数据会被呈现以及它们如何被格式化是很清楚的。

# 基本用法

你可以定义一个字典或者 `fields` 的 OrderedDict 类型，OrderedDict 类型是指键名是要呈现的对象的属性或键的名称，键值是一个类，该类格式化和返回的该字段的值。这个例子有三个字段，两个是字符串（Strings）以及一个是日期时间（DateTime），格式为 RFC 822 日期字符串（同样也支持 ISO 8601）

```
from flask.ext.restful import Resource, fields, marshal_with

resource_fields = {
    'name': fields.String,
    'address': fields.String,
    'date_updated': fields.DateTime(dt_format='rfc822'),
}

class Todo(Resource):
    @marshal_with(resource_fields, envelope='resource')
    def get(self, **kwargs):
        return db_get_todo()  # Some function that queries the db
```

这个例子假设你有一个自定义的数据库对象（ `todo` ），它具有属性： `name` ， `address` ，以及 `date_updated` 。该对象上任何其它的属性可以被认为是私有的不会在输出中呈现出来。一个可选的 `envelope` 关键字参数被指定为封装结果输出。

装饰器 `marshal_with` 是真正接受你的数据对象并且过滤字段。 `marshal_with` 能够在单个对象，字典，或者列表对象上工作。

注意：marshal_with 是一个很便捷的装饰器，在功能上等效于如下的 `return marshal(db_get_todo(), resource_fields), 200` 。这个明确的表达式能用于返回 200 以及其它的 HTTP 状态码作为成功响应（错误响应见 `abort` ）。

# 重命名属性

很多时候你面向公众的字段名称是不同于内部的属性名。使用 `attribute` 可以配置这种映射。

```
fields = {
    'name': fields.String(attribute='private_name'),
    'address': fields.String,
}
```

lambda 也能在 `attribute` 中使用

```
fields = {
    'name': fields.String(attribute=lambda x: x._private_name),
    'address': fields.String,
}
```

# 默认值

如果由于某种原因你的数据对象中并没有你定义的字段列表中的属性，你可以指定一个默认值而不是返回 `None` 。

```
fields = {
    'name': fields.String(default='Anonymous User'),
    'address': fields.String,
}
```

# 自定义字段&多个值

有时候你有你自己定义格式的需求。你可以继承 `fields.Raw` 类并且实现格式化函数。当一个属性存储多条信息的时候是特别有用的。例如，一个位域（bit-field）各位代表不同的值。你可以使用 `fields` 复用一个单一的属性到多个输出值（一个属性在不同情况下输出不同的结果）。

这个例子假设在 `flags` 属性的第一位标志着一个“正常”或者“迫切”项，第二位标志着“读”与“未读”。这些项可能很容易存储在一个位字段，但是可读性不高。转换它们使得具有良好的可读性是很容易的。

```
class UrgentItem(fields.Raw):
    def format(self, value):
        return "Urgent" if value & 0x01 else "Normal"

class UnreadItem(fields.Raw):
    def format(self, value):
        return "Unread" if value & 0x02 else "Read"

fields = {
    'name': fields.String,
    'priority': UrgentItem(attribute='flags'),
    'status': UnreadItem(attribute='flags'),
}
```

# Url & 其它具体字段

Flask-RESTful 包含一个特别的字段， `fields.Url` ，即为所请求的资源合成一个 uri。这也是一个好示例，它展示了如何添加并不真正在你的数据对象中存在的数据到你的响应中。

```
class RandomNumber(fields.Raw):
    def output(self, key, obj):
        return random.random()

fields = {
    'name': fields.String,
    # todo_resource is the endpoint name when you called api.add_resource()
    'uri': fields.Url('todo_resource'),
    'random': RandomNumber,
}
```

默认情况下，`fields.Url` 返回一个相对的 uri。为了生成包含协议（scheme），主机名以及端口的绝对 uri，需要在字段声明的时候传入 `absolute=True`。传入 `scheme` 关键字参数可以覆盖默认的协议（scheme）：

```
fields = {
    'uri': fields.Url('todo_resource', absolute=True)
    'https_uri': fields.Url('todo_resource', absolute=True, scheme='https')
}
```

# 复杂结构

你可以有一个扁平的结构，marshal_with 将会把它转变为一个嵌套结构

```
>>> from flask.ext.restful import fields, marshal
>>> import json
>>>
>>> resource_fields = {'name': fields.String}
>>> resource_fields['address'] = {}
>>> resource_fields['address']['line 1'] = fields.String(attribute='addr1')
>>> resource_fields['address']['line 2'] = fields.String(attribute='addr2')
>>> resource_fields['address']['city'] = fields.String
>>> resource_fields['address']['state'] = fields.String
>>> resource_fields['address']['zip'] = fields.String
>>> data = {'name': 'bob', 'addr1': '123 fake street', 'addr2': '', 'city': 'New York', '
>>> json.dumps(marshal(data, resource_fields))
'{"name": "bob", "address": {"line 1": "123 fake street", "line 2": "", "state": "NY", "z
```

注意：address 字段并不真正地存在于数据对象中，但是任何一个子字段（sub-fields）可以直接地访问对象的属性，就像没有嵌套一样。

# 列表字段

你也可以把字段解组（unmarshal）成列表

```
>>> from flask.ext.restful import fields, marshal
>>> import json
>>>
>>> resource_fields = {'name': fields.String, 'first_names': fields.List(fields.String)}
>>> data = {'name': 'Bougnazal', 'first_names' : ['Emile', 'Raoul']}
>>> json.dumps(marshal(data, resource_fields))
>>> '{"first_names": ["Emile", "Raoul"], "name": "Bougnazal"}'
```

# 高级：嵌套字段

尽管使用字典套入字段能够使得一个扁平的数据对象变成一个嵌套的响应，你可以使用 `Nested` 解组（unmarshal）嵌套数据结构并且合适地呈现它们。

```
>>> from flask.ext.restful import fields, marshal
>>> import json
>>>
>>> address_fields = {}
>>> address_fields['line 1'] = fields.String(attribute='addr1')
>>> address_fields['line 2'] = fields.String(attribute='addr2')
>>> address_fields['city'] = fields.String(attribute='city')
>>> address_fields['state'] = fields.String(attribute='state')
>>> address_fields['zip'] = fields.String(attribute='zip')
>>>
>>> resource_fields = {}
>>> resource_fields['name'] = fields.String
>>> resource_fields['billing_address'] = fields.Nested(address_fields)
>>> resource_fields['shipping_address'] = fields.Nested(address_fields)
>>> address1 = {'addr1': '123 fake street', 'city': 'New York', 'state': 'NY', 'zip': '10
>>> address2 = {'addr1': '555 nowhere', 'city': 'New York', 'state': 'NY', 'zip': '10468'
>>> data = { 'name': 'bob', 'billing_address': address1, 'shipping_address': address2}
>>>
>>> json.dumps(marshal_with(data, resource_fields))
'{"billing_address": {"line 1": "123 fake street", "line 2": null, "state": "NY", "zip":
```

此示例使用两个嵌套字段。 `Nested` 构造函数把字段的字典作为子字段（sub-fields）来呈现。使用 `Nested` 和之前例子中的嵌套字典之间的重要区别就是属性的上下文。在本例中 “billing_address” 是一个具有自己字段的复杂的对象，传递给嵌套字段的上下文是子对象（sub-object），而不是原来的“数据”对象。换句话说， `data.billing_address.addr1` 是在这里的范围（译者：这里是直译），然而在之前例子中的 `data.addr1` 是位置属性。记住：嵌套和列表对象创建一个新属性的范围。

# 扩展 Flask-RESTful

我们认识到每一个人在 REST 框架上有着不同的需求。Flask-RESTful 试图尽可能的灵活，但是有时候你可能会发现内置的功能不足够满足你的需求。Flask-RESTful 有几个不同的扩展点，这些扩展在这种情况下会有帮助。

# 内容协商

开箱即用，Flask-RESTful 仅配置为支持 JSON。我们做出这个决定是为了给 API 维护者完全控制 API 格式支持，因此一年来的路上，你不必支持那些使用 API 且用 CSV 表示的人们，甚至你都不知道他们的存在。要添加其它的 mediatypes 到你的 API 中，你需要在 `Api` 对象中声明你支持的表示。

```
app = Flask(__name__)
api = restful.Api(app)

@api.representation('application/json')
def output_json(data, code, headers=None):
    resp = make_response(json.dumps(data), code)
    resp.headers.extend(headers or {})
    return resp
```

这些表示函数必须返回一个 Flask `Response` 对象。

# 自定义字段 & 输入

一种最常见的 Flask-RESTful 附件功能就是基于你自己数据类型的数据来定义自定义的类型或者字段。

## 字段

自定义输出字段让你无需直接修改内部对象执行自己的输出格式。所有你必须做的就是继承 `Raw` 并且实现 `format()` 方法:

```
class AllCapsString(fields.Raw):
    def format(self, value):
        return value.upper()

# example usage
fields = {
    'name': fields.String,
    'all_caps_name': AllCapsString(attribute=name),
}
```

## 输入

对于解析参数，你可能要执行自定义验证。创建你自己的输入类型让你轻松地扩展请求解析。

```
def odd_number(value):
    if value % 2 == 0:
        raise ValueError("Value is not odd")

    return value
```

请求解析器在你想要在错误消息中引用名称的情况下将也会允许你访问参数的名称。

```
def odd_number(value, name):
    if value % 2 == 0:
        raise ValueError("The parameter '{}' is not odd. You gave us the value: {}".forma

    return value
```

你还可以将公开的参数转换为内部表示：

```
# maps the strings to their internal integer representation
# 'init' => 0
# 'in-progress' => 1
# 'completed' => 2

def task_status(value):
    statuses = [u"init", u"in-progress", u"completed"]
    return statuses.index(value)
```

然后你可以在你的 RequestParser 中使用这些自定义输入类型：

```
parser = reqparse.RequestParser()
parser.add_argument('OddNumber', type=odd_number)
parser.add_argument('Status', type=task_status)
args = parser.parse_args()
```

# 响应格式

为了支持其它的表示（像 XML,CSV,HTML），你可以使用 `representation()` 装饰器。你需要在你的 API 中引用它。

```
api = restful.Api(app)

@api.representation('text/csv')
def output_csv(data, code, headers=None):
    pass
    # implement csv output!
```

这些输出函数有三个参数，`data`，`code`，以及 `headers`。

`data` 是你从你的资源方法返回的对象，`code` 是预计的 HTTP 状态码，`headers` 是设置在响应中任意的 HTTP 头。你的输出函数应该返回一个 Flask 响应对象。

```
def output_json(data, code, headers=None):
    """Makes a Flask response with a JSON encoded body"""
    resp = make_response(json.dumps(data), code)
    resp.headers.extend(headers or {})

    return resp
```

另外一种实现这一点的就是继承 `Api` 类并且提供你自己输出函数。

```
class Api(restful.Api):
    def __init__(self, *args, **kwargs):
        super(Api, self).__init__(*args, **kwargs)
        self.representations = {
            'application/xml': output_xml,
            'text/html': output_html,
            'text/csv': output_csv,
            'application/json': output_json,
        }
```

# 资源方法装饰器

`Resource()` 有一个叫做 method_decorators 的属性。你可以继承 Resource 并且添加你自己的装饰器，该装饰器将会被添加到资源里面所有 `method` 函数。举例来说，如果你想要为每一个请求建立自定义认证。

```
def authenticate(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if not getattr(func, 'authenticated', True):
            return func(*args, **kwargs)

        acct = basic_authentication()  # custom account lookup function

        if acct:
            return func(*args, **kwargs)

        restful.abort(401)
    return wrapper

class Resource(restful.Resource):
    method_decorators = [authenticate]   # applies to all inherited resources
```

由于 Flask-RESTful Resources 实际上是 Flask 视图对象，你也可以使用标准的 flask 视图装饰器。

# 自定义错误处理器

错误处理是一个很棘手的问题。你的 Flask 应用可能身兼数职，然而你要以正确的内容类型以及错误语法处理所有的 Flask-RESTful 错误。

Flask-RESTful 在 Flask-RESTful 路由上发生任何一个 400 或者 500 错误的时候调用 `handle_error()` 函数，不会干扰到其它的路由。你可能需要你的应用程序在 404 Not Found 错误上返回一个携带正确媒体类型（介质类型）的错误信息；在这种情况下，使用 `Api` 构造函数的 `catch_all_404s` 参数。

```
app = Flask(__name__)
api = flask_restful.Api(app, catch_all_404s=True)
```

Flask-RESTful 会处理除了自己路由上的错误还有应用程序上所有的 404 错误。

有时候你想在发生错误的时候做一些特别的东西 - 记录到文件，发送邮件，等等。使用 `got_request_exception()` 方法把自定义错误处理加入到异常。

```
def log_exception(sender, exception, **extra):
    """ Log an exception to our logging framework """
    sender.logger.debug('Got exception during processing: %s', exception)

from flask import got_request_exception
got_request_exception.connect(log_exception, app)
```

## 定义自定义错误消息

在一个请求期间遇到某些错误的时候，你可能想返回一个特定的消息以及/或者状态码。你可以告诉 Flask-RESTful 你要如何处理每一个错误/异常，因此你不必在你的 API 代码中编写 try/except 代码块。

```
errors = {
    'UserAlreadyExistsError': {
        'message': "A user with that username already exists.",
        'status': 409,
    },
    'ResourceDoesNotExist': {
        'message': "A resource with that ID no longer exists.",
        'status': 410,
        'extra': "Any extra information you want.",
    },
}
```

包含 `'status'` 键可以设置响应的状态码。如果没有指定的话，默认是 500.

一旦你的 `errors` 字典定义，简单地把它传给 `Api` 构造函数。

```
app = Flask(__name__)
api = flask_restful.Api(app, errors=errors)
```

# 中高级用法

本页涉及构建一个稍微复杂的 Flask-RESTful 应用程序，该应用程序将会覆盖到一些最佳练习当你建立一个真实世界的基于 Flask-RESTful 的 API。快速入门 章节适用于开始你的第一个 Flask-RESTful 应用程序，因此如果你是 Flask-RESTful 的新用户，最好查阅该章节。

## 项目结构

有许多不同的方式来组织你的 Flask-RESTful 应用程序，但是这里我们描述了一个在大型的应用程序中能够很好地扩展并且维持一个不错的文件组织。

最基本的思路就是把你的应用程序分为三个主要部分。路由，资源，以及任何公共的基础部分。

下面是目录结构的一个例子：

```
myapi/
    __init__.py
    app.py          # this file contains your app and routes
    resources/
        __init__.py
        foo.py      # contains logic for /Foo
        bar.py      # contains logic for /Bar
    common/
        __init__.py
        util.py     # just some common infrastructure
```

common 文件夹可能只包含一组辅助函数以满足你的应用程序公共的需求。例如，它也可能包含任何自定义输入/输出类型。

在 resource 文件夹中，你只有资源对象。因此这里就是 foo.py 可能的样子：

```
from flask.ext import restful

class Foo(restful.Resource):
    def get(self):
        pass
    def post(self):
        pass
```

app.py 中的配置就像这样：

```
from flask import Flask
from flask.ext import restful
from myapi.resources.foo import Foo
from myapi.resources.bar import Bar
from myapi.resources.baz import Baz

app = Flask(__name__)
api = restful.Api(app)

api.add_resource(Foo, '/Foo', '/Foo/<str:id>')
api.add_resource(Bar, '/Bar', '/Bar/<str:id>')
api.add_resource(Baz, '/Baz', '/Baz/<str:id>')
```

因为你可能编写一个特别大型或者复杂的 API，这个文件里面会有一个所有路由以及资源的复杂列表。你也可以使用这个文件来设置任何的配置值（before_request，after_request）。基本上，这个文件配置你整个 API。

# 完整的参数解析示例

在文档的其它地方，我们已经详细地介绍了如何使用 reqparse 的例子。这里我们将设置一个有多个输入参数的资源。我们将定义一个名为 “User” 的资源。

```
from flask.ext import restful
from flask.ext.restful import fields, marshal_with, reqparse

def email(email_str):
    """ return True if email_str is a valid email """
    if valid_email(email):
        return True
    else:
        raise ValidationError("{} is not a valid email")

post_parser = reqparse.RequestParser()
post_parser.add_argument(
    'username', dest='username',
    type=str, location='form',
    required=True, help='The user\'s username',
)
post_parser.add_argument(
    'email', dest='email',
    type=email, location='form',
    required=True, help='The user\'s email',
)
post_parser.add_argument(
    'user_priority', dest='user_priority',
    type=int, location='form',
    default=1, choices=range(5), help='The user\'s priority',
)

user_fields = {
    'id': fields.Integer,
    'username': fields.String,
    'email': fields.String,
    'user_priority': fields.Integer,
    'custom_greeting': fields.FormattedString('Hey there {username}!'),
    'date_created': fields.DateTime,
    'date_updated': fields.DateTime,
    'links': fields.Nested({
        'friends': fields.Url('/Users/{id}/Friends'),
        'posts': fields.Url('Users/{id}/Posts'),
    }),
}

class User(restful.Resource):

    @marshal_with(user_fields)
    def post(self):
        args = post_parser.parse_args()
        user = create_user(args.username, args.email, args.user_priority)
        return user

    @marshal_with(user_fields)
    def get(self, id):
        args = get_parser.parse_args()
        user = fetch_user(id)
        return user
```

正如你所看到的，我们创建一个 `post_parser` 专门用来处理解析 POST 请求携带的参数。让我们逐步介绍每一个定义的参数。

```
post_parser.add_argument(
    'username', dest='username',
    type=str, location='form',
    required=True, help='The user\'s username',
)
```

`username` 字段是所有参数中最为普通的。它从 POST 数据中获取一个字符串并且把它转换成一个字符串类型。该参数是必须得（`required=True`），这就意味着如果不提供改参数，Flask-RESTful 会自动地返回一个消息是'用户名字段是必须'的 400 错误。

```
post_parser.add_argument(
    'email', dest='email',
    type=email, location='form',
    required=True, help='The user\'s email',
)
```

`email` 字段是一个自定义的 `email` 类型。在最前面几行中我们定义了一个 `email` 函数，它接受一个字符串，如果该字符串类型合法的话返回 True，否则会引起一个 `ValidationError` 异常，该异常明确表示 email 类型不合法。

```
post_parser.add_argument(
    'user_priority', dest='user_priority',
    type=int, location='form',
    default=1, choices=range(5), help='The user\'s priority',
)
```

`user_priority` 类型充分利用了 `choices` 参数。这就意味着如果提供的 `user_priority` 值不落在由 `choices` 参数指定的范围内的话，Flask-RESTful 会自动地以 400 状态码以及一个描述性的错误消息响应。

下面该讨论到输入了。我们也在 `user_fields` 字典中定义了一些有趣的字段类型用来展示一些特殊的类型。

```
user_fields = {
    'id': fields.Integer,
    'username': fields.String,
    'email': fields.String,
    'user_priority': fields.Integer,
    'custom_greeting': fields.FormattedString('Hey there {username}!'),
    'date_created': fields.DateTime,
    'date_updated': fields.DateTime,
    'links': fields.Nested({
        'friends': fields.Url('/Users/{id}/Friends', absolute=True),
        'posts': fields.Url('Users/{id}/Posts', absolute=True),
    }),
}
```

首先，存在一个 `fields.FormattedString`。

```
'custom_greeting': fields.FormattedString('Hey there {username}!'),
```

此字段主要用于篡改响应中的值到其它的值。在这种情况下，`custom_greeting` 将总是包含从 `username` 字段返回的值。

下一步，检查 `fields.Nested`。

```
'links': fields.Nested({
    'friends': fields.Url('/Users/{id}/Friends', absolute=True),
    'posts': fields.Url('Users/{id}/Posts', absolute=True),
}),
```

此字段是用于在响应中创建子对象。在这种情况下，我们要创建一个包含相关对象 urls 的 `links` 子对象。注意这里我们是使用了 `fields.Nested` 。

最后，我们使用了 `fields.Url` 字段类型。

```
'friends': fields.Url('/Users/{id}/Friends', absolute=True),
'posts': fields.Url('Users/{id}/Posts', absolute=True),
```

它接受一个字符串作为参数，它能以我们上面提到的 `fields.FormattedString` 同样的方式被格式化。传入 `absolute=True` 确保生成的 Urls 包含主机名。

# API 文档

`flask.ext.restful.marshal(data, fields, envelope=None)`

Takes raw data (in the form of a dict, list, object) and a dict of fields to output and filters the data based on those fields.

Parameters:

- **data** – the actual object(s) from which the fields are taken from
- **fields** – a dict of whose keys will make up the final serialized response output
- **envelope** – optional key that will be used to envelop the serialized response

```
>>> from flask.ext.restful import fields, marshal
>>> data = { 'a': 100, 'b': 'foo' }
>>> mfields = { 'a': fields.Raw }
```

```
>>> marshal(data, mfields)
OrderedDict([('a', 100)])
```

```
>>> marshal(data, mfields, envelope='data')
OrderedDict([('data', OrderedDict([('a', 100)]))])
```

`flask.ext.restful.marshal_with(fields, envelope=None)`

A decorator that apply marshalling to the return values of your methods.

```
>>> from flask.ext.restful import fields, marshal_with
>>> mfields = { 'a': fields.Raw }
>>> @marshal_with(mfields)
... def get():
...     return { 'a': 100, 'b': 'foo' }
...
...
>>> get()
OrderedDict([('a', 100)])
```

```
>>> @marshal_with(mfields, envelope='data')
... def get():
...     return { 'a': 100, 'b': 'foo' }
...
...
>>> get()
OrderedDict([('data', OrderedDict([('a', 100)]))])
```

see `flask.ext.restful.marshal()`

`flask.ext.restful.marshal_with_field(field)`

A decorator that formats the return values of your methods with a single field.

```
>>> from flask.ext.restful import marshal_with_field, fields
>>> @marshal_with_field(fields.List(fields.Integer))
... def get():
...     return ['1', 2, 3.0]
...
>>> get()
[1, 2, 3]
```

see `flask.ext.restful.marshal_with()`

`flask.ext.restful.abort(http_status_code, **kwargs)`

Raise a HTTPException for the given http_status_code. Attach any keyword arguments to the exception for later processing.

# Api

`class flask.ext.restful.Api(app=None, prefix='', default_mediatype='application/json', dec`

The main entry point for the application. You need to initialize it with a Flask Application:

```
>>> app = Flask(__name__)
>>> api = restful.Api(app)
```

Alternatively, you can use `init_app()` to set the Flask application after it has been constructed.

Parameters:

- **app** (*flask.Flask*) – the Flask application object
- **prefix** (*str*) – Prefix all routes with a value, eg v1 or 2010-04-01
- **default_mediatype** (*str*) – The default media type to return
- **decorators** (*list*) – Decorators to attach to every resource
- **catch_all_404s** (*bool*) – Use `handle_error()` to handle 404 errors throughout your app
- **url_part_order** – A string that controls the order that the pieces of the url are concatenated when the full url is constructed. 'b' is the blueprint (or blueprint registration) prefix, 'a' is the api prefix, and 'e' is the path component the endpoint is added with
- **errors** () – A dictionary to define a custom response for each exception or error raised during a request

`add_resource(resource, *urls, **kwargs)`

Adds a resource to the api.

Parameters:

- **resource** ( `Resource` ) – the class name of your resource
- **urls** (*str*) – one or more url routes to match for the resource, standard flask routing rules apply. Any url variables will be passed to the resource method as args.
- **endpoint** (*str*) – endpoint name (defaults to `Resource.__name__.lower()` Can be used to reference this route in `fields.Url` fields

Additional keyword arguments not specified above will be passed as-is to `flask.Flask.add_url_rule()` .

Examples:

```
api.add_resource(HelloWorld, '/', '/hello')
api.add_resource(Foo, '/foo', endpoint="foo")
api.add_resource(FooSpecial, '/special/foo', endpoint="foo")
```

`error_router(original_handler, e)`

This function decides whether the error occured in a flask-restful endpoint or not. If it happened in a flask-restful endpoint, our handler will be dispatched. If it happened in an unrelated view, the app's original error handler will be dispatched. In the event that the error occurred in a flask-restful endpoint but the local handler can't resolve the situation, the router will fall back onto the original_handler as last resort.

Parameters:

- **original_handler** (*function*) – the original Flask error handler for the app
- **e** (*Exception*) – the exception raised while handling the request

`handle_error(e)`

Error handler for the API transforms a raised exception into a Flask response, with the appropriate HTTP status code and body.

Parameters: **e** (*Exception*) – the raised Exception object

`init_app(app)`

Initialize this class with the given `flask.Flask` application or `flask.Blueprint` object.

Parameters: **app** (*flask.Blueprint*) – the Flask application or blueprint object

Examples:

```
api = Api()
api.add_resource(...)
api.init_app(app)
```

`make_response(data, *args, **kwargs)`

Looks up the representation transformer for the requested media type, invoking the transformer to create a response object. This defaults to (application/json) if no transformer is found for the requested mediatype.

Parameters: **data** – Python object containing response data to be transformed

`mediatypes()`

Returns a list of requested mediatypes sent in the Accept header

`mediatypes_method()`

Return a method that returns a list of mediatypes

`output(resource)`

Wraps a resource (as a flask view function), for cases where the resource does not directly return a response object

Parameters: **resource** – The resource as a flask view function

`owns_endpoint(endpoint)`

Tests if an endpoint name (not path) belongs to this Api. Takes in to account the Blueprint name part of the endpoint name.

Parameters: **endpoint** – The name of the endpoint being checked

Returns: bool

`representation(mediatype)`

Allows additional representation transformers to be declared for the api. Transformers are functions that must be decorated with this method, passing the mediatype the transformer represents. Three arguments are passed to the transformer:

- The data to be represented in the response body
- The http status code
- A dictionary of headers

The transformer should convert the data appropriately for the mediatype and return a Flask response object.

Ex:

```
@api.representation('application/xml')
def xml(data, code, headers):
    resp = make_response(convert_data_to_xml(data), code)
    resp.headers.extend(headers)
    return resp
```

`resource(*urls, **kwargs)`

Wraps a `Resource` class, adding it to the api. Parameters are the same as `add_resource()`.

Example:

```
app = Flask(__name__)
api = restful.Api(app)

@api.resource('/foo')
class Foo(Resource):
    def get(self):
        return 'Hello, World!'
```

`unauthorized(response)`

Given a response, change it to ask for credentials

`url_for(resource, **values)`

Generates a URL to the given resource.

`class flask.ext.restful.Resource`

Represents an abstract RESTful resource. Concrete resources should extend from this class and expose methods for each supported HTTP method. If a resource is invoked with an unsupported HTTP method, the API will return a response with status 405 Method Not Allowed. Otherwise the appropriate method is called and passed all arguments from the url rule used when adding the resource to an Api instance. See `add_resource()` for details.

# ReqParse

```
class reqparse.RequestParser(argument_class=<class 'reqparse.Argument'>, namespace_class=<
```

Enables adding and parsing of multiple arguments in the context of a single request. Ex:

```
from flask import request

parser = RequestParser()
parser.add_argument('foo')
parser.add_argument('int_bar', type=int)
args = parser.parse_args()
```

```
add_argument(*args, **kwargs)
```

Adds an argument to be parsed.

Accepts either a single instance of Argument or arguments to be passed into `Argument` 's constructor.

See `Argument` 's constructor for documentation on the available options.

```
copy()
```

Creates a copy of this RequestParser with the same set of arguments

```
parse_args(req=None, strict=False)
```

Parse all arguments from the provided request and return the results as a Namespace

Parameters: **strict** – if req includes args not in parser, throw 400 BadRequest exception

```
remove_argument(name)
```

Remove the argument matching the given name.

```
replace_argument(name, *args, **kwargs)
```

Replace the argument matching the given name with a new version.

```
class reqparse.Argument(name, default=None, dest=None, required=False, ignore=False, type=
```

Parameters:

- **name** – Either a name or a list of option strings, e.g. foo or -f, –foo.
- **default** – The value produced if the argument is absent from the request.
- **dest** – The name of the attribute to be added to the object returned by `parse_args()` .
- **required** (*bool*) – Whether or not the argument may be omitted (optionals only).
- **action** – The basic type of action to be taken when this argument is encountered in the request. Valid options are “store” and “append”.
- **ignore** – Whether to ignore cases where the argument fails type conversion
- **type** – The type to which the request argument should be converted. If a type raises a ValidationError, the message in the error will be returned in the response. Defaults to `unicode` in python2 and `str` in python3.

- **location** – The attributes of the `flask.Request` object to source the arguments from (ex: headers, args, etc.), can be an iterator. The last item listed takes precedence in the result set.
- **choices** – A container of the allowable values for the argument.
- **help** – A brief description of the argument, returned in the response when the argument is invalid with the name of the argument and the message passed to a ValidationError raised by a type converter.
- **case_sensitive** (*bool*) – Whether the arguments in the request are case sensitive or not
- **store_missing** (*bool*) – Whether the arguments default value should be stored if the argument is missing from the request.

`__init__(name, default=None, dest=None, required=False, ignore=False, type=<function <lamb`

`handle_validation_error(error)`

Called when an error is raised while parsing. Aborts the request with a 400 status and an error message

Parameters: **error** – the error that was raised

`parse(request)`

Parses argument value(s) from the request, converting according to the argument's type.

Parameters: **request** – The flask request object to parse arguments from

`source(request)`

Pulls values off the request in the provided location :param request: The flask request object to parse arguments from

# Fields

`class fields.String(default=None, attribute=None)`

Marshal a value as a string. Uses `six.text_type` so values will be converted to `unicode` in python2 and `str` in python3.

`format(value)`

`class fields.Url(endpoint=None, absolute=False, scheme=None)`

FormattedString is used to interpolate other values from the response into this field. The syntax for the source string is the same as the string `format` method from the python stdlib.

Ex:

```
fields = {
    'name': fields.String,
    'greeting': fields.FormattedString("Hello {name}")
}
data = {
    'name': 'Doug',
}
marshal(data, fields)
```

`output(key, obj)`

`class fields.Url(endpoint=None, absolute=False, scheme=None)`

A string representation of a Url

`output(key, obj)`

`class fields.DateTime(dt_format='rfc822', **kwargs)`

Return a formatted datetime string in UTC. Supported formats are RFC 822 and ISO 8601.

Parameters: **dt_format** (*str*) – `'rfc822'` or `'iso8601'`

`format(value)`

`class fields.Float(default=None, attribute=None)`

A double as IEEE-754 double precision. ex : 3.141592653589793 3.1415926535897933e-06 3.141592653589793e+24 nan inf -inf

`format(value)`

`class fields.Integer(default=0, **kwargs)`

Field for outputting an integer value.

Parameters:

- **default** (*int*) – The default value for the field, if no value is specified.
- **attribute** – If the public facing value differs from the internal value, use this to retrieve a different attribute from the response than the publicly named value.

`format(value)`

`class fields.Arbitrary(default=None, attribute=None)`

A floating point number with an arbitrary precision

ex: 634271127864378216478362784632784678324.23432

`format(value)`

`class fields.Nested(nested, allow_null=False, **kwargs)`

Allows you to nest one set of fields inside another. See 高级：嵌套字段 for more information

Parameters:

- **nested** (*dict*) – The dictionary to nest
- **allow_null** (*bool*) – Whether to return None instead of a dictionary with null keys, if a nested dictionary has all-null keys
- ****kwargs** – if `default` keyword argument is present, a nested dictionary will be marshaled as its value if nested dictionary is all-null keys (e.g. lets you return an empty JSON object instead of null)

:keyword default

`output(key, obj)`

`class fields.List(cls_or_instance, **kwargs)`

Field for marshalling lists of other fields.

See 列表字段 for more information.

Parameters: **cls_or_instance** – The field type the list will contain.

`format(value)`

`output(key, data)`

`class fields.Raw(default=None, attribute=None)`

Raw provides a base field class from which others should extend. It applies no formatting by default, and should only be used in cases where data does not need to be formatted before being serialized. Fields should throw a MarshallingException in case of parsing problem.

`format(value)`

Formats a field’s value. No-op by default - field classes that modify how the value of existing object keys should be presented should override this and apply the appropriate formatting.

Parameters: **value** – The value to format

Raises MarshallingException:

In case of formatting problem

Ex:

```
class TitleCase(Raw):
    def format(self, value):
        return unicode(value).title()
```

`output(key, obj)`

Pulls the value for the given key from the object, applies the field's formatting and returns the result. If the key is not found in the object, returns the default value. Field classes that create values which do not require the existence of the key in the object should override this and return the desired value.

Raises MarshallingException:

In case of formatting problem

`class fields.Boolean(default=None, attribute=None)`

Field for outputting a boolean value.

Empty collections such as `""` , `{}` , `[]` , etc. will be converted to `False` .

`format(value)`

`class fields.Fixed(decimals=5, **kwargs)`

A decimal number with a fixed precision.

`format(value)`

`fields.Price`

alias of `Fixed`

# Inputs

`flask.ext.restful.inputs.url(value)`

Validate a URL.

Parameters: **value** (*string*) – The URL to validate

Returns: The URL if valid.

Raises: ValueError

`flask.ext.restful.inputs.regex`

`flask.ext.restful.inputs.date(value)`

Parse a valid looking date in the format YYYY-mm-dd

`flask.ext.restful.inputs.iso8601interval(value, argument='argument')`

Parses ISO 8601-formatted datetime intervals into tuples of datetimes.

Accepts both a single date(time) or a full interval using either start/end or start/duration notation, with the following behavior:

- Intervals are defined as inclusive start, exclusive end

- Single datetimes are translated into the interval spanning the largest resolution not specified in the input value, up to the day.
- The smallest accepted resolution is 1 second.
- All timezones are accepted as values; returned datetimes are localized to UTC. Naive inputs and date inputs will are assumed UTC.

Examples:

```
"2013-01-01" -> datetime(2013, 1, 1), datetime(2013, 1, 2)
"2013-01-01T12" -> datetime(2013, 1, 1, 12), datetime(2013, 1, 1, 13)
"2013-01-01/2013-02-28" -> datetime(2013, 1, 1), datetime(2013, 2, 28)
"2013-01-01/P3D" -> datetime(2013, 1, 1), datetime(2013, 1, 4)
"2013-01-01T12:00/PT30M" -> datetime(2013, 1, 1, 12), datetime(2013, 1, 1, 12, 30)
"2013-01-01T06:00/2013-01-01T12:00" -> datetime(2013, 1, 1, 6), datetime(2013, 1, 1, 12)
```

Parameters: **value** (*str*) – The ISO8601 date time as a string

Returns: Two UTC datetimes, the start and the end of the specified interval

Return type: A tuple (datetime, datetime)

Raises: ValueError, if the interval is invalid.

`flask.ext.restful.inputs.natural(value, argument='argument')`

Restrict input type to the natural numbers (0, 1, 2, 3...)

`flask.ext.restful.inputs.boolean(value)`

Parse the string “true” or “false” as a boolean (case insensitive). Also accepts “1” and “0” as True/False (respectively). If the input is from the request JSON body, the type is already a native python boolean, and will be passed through without further parsing.

`flask.ext.restful.inputs.rfc822(dt)`

Turn a datetime object into a formatted date.

Example:

```
inputs.rfc822(datetime(2011, 1, 1)) => "Sat, 01 Jan 2011 00:00:00 -0000"
```

Parameters: **dt** (*datetime*) – The datetime to transform

Returns: A RFC 822 formatted date string

# 运行测试

`Makefile` 文件中照顾到搭建一个 virtualenv 为运行测试。所有你需要做的就是运行:

```
$ make test
```

要更改用于运行测试的 Python 版本（默认是 Python 2.7），修改上面 `Makefile` 文件中的 `PYTHON_MAJOR` 和 `PYTHON_MINOR` 变量。

你可以在所有支持的版本上运行测试:

```
$ make test-all
```

单个的测试可以使用如下格式的命令运行:

```
nosetests <filename>:ClassName.func_name
```

例如:

```
$ source env/bin/activate
$ nosetests tests/test_reqparse.py:ReqParseTestCase.test_parse_choices_insensitive
```

另外，提交你的更改到 Github 中你的分支上，Travis 将会自动地为你的分支运行测试。

也提供了一个 Tox 配置文件，因此你可以在本地在多个 python 版本（2.6，2.7，3.3，和 3.4）上测试

```
$ tox
```

# 使用 Flask 设计 RESTful APIs

翻译者注：本系列的原文名为：Designing a RESTful API with Python and Flask ， 作者是 **Miguel Grinberg** 。

# 使用 Python 和 Flask 设计 RESTful API

近些年来 REST (REpresentational State Transfer) 已经变成了 web services 和 web APIs 的标配。

在本文中我将向你展示如何简单地使用 Python 和 Flask 框架来创建一个 RESTful 的 web service。

## 什么是 REST？

六条设计规范定义了一个 REST 系统的特点:

- 客户端-服务器: 客户端和服务器之间隔离，服务器提供服务，客户端进行消费。
- 无状态: 从客户端到服务器的每个请求都必须包含理解请求所必需的信息。换句话说， 服务器不会存储客户端上一次请求的信息用来给下一次使用。
- 可缓存: 服务器必须明示客户端请求能否缓存。
- 分层系统: 客户端和服务器之间的通信应该以一种标准的方式，就是中间层代替服务器做出响应的时候，客户端不需要做任何变动。
- 统一的接口: 服务器和客户端的通信方法必须是统一的。
- 按需编码: 服务器可以提供可执行代码或脚本，为客户端在它们的环境中执行。这个约束是唯一一个是可选的。

## 什么是一个 RESTful 的 web service？

REST 架构的最初目的是适应万维网的 HTTP 协议。

RESTful web services 概念的核心就是“资源”。 资源可以用 URI 来表示。客户端使用 HTTP 协议定义的方法来发送请求到这些 URIs，当然可能会导致这些被访问的”资源“状态的改变。

HTTP 标准的方法有如下:

```
==========  =====================  ==================================
HTTP 方法   行为                   示例
==========  =====================  ==================================
GET         获取资源的信息         http://example.com/api/orders
GET         获取某个特定资源的信息 http://example.com/api/orders/123
POST        创建新资源             http://example.com/api/orders
PUT         更新资源               http://example.com/api/orders/123
DELETE      删除资源               http://example.com/api/orders/123
==========  ====================== ==================================
```

REST 设计不需要特定的数据格式。在请求中数据可以以 JSON 形式, 或者有时候作为 url 中查询参数项。

# 设计一个简单的 web service

坚持 REST 的准则设计一个 web service 或者 API 的任务就变成一个标识资源被展示出来以及它们是怎样受不同的请求方法影响的练习。

比如说，我们要编写一个待办事项应用程序而且我们想要为它设计一个 web service。要做的第一件事情就是决定用什么样的根 URL 来访问该服务。例如，我们可以通过这个来访问:

http://[hostname]/todo/api/v1.0/

在这里我已经决定在 URL 中包含应用的名称以及 API 的版本号。在 URL 中包含应用名称有助于提供一个命名空间以便区分同一系统上的其它服务。在 URL 中包含版本号能够帮助以后的更新，如果新版本中存在新的和潜在不兼容的功能，可以不影响依赖于较旧的功能的应用程序。

下一步骤就是选择将由该服务暴露(展示)的资源。这是一个十分简单地应用，我们只有任务，因此在我们待办事项中唯一的资源就是任务。

我们的任务资源将要使用 HTTP 方法如下:

```
==========  ===============================================  ============================
HTTP 方法   URL                                              动作
==========  ===============================================  ============================
GET         http://[hostname]/todo/api/v1.0/tasks            检索任务列表
GET         http://[hostname]/todo/api/v1.0/tasks/[task_id]  检索某个任务
POST        http://[hostname]/todo/api/v1.0/tasks            创建新任务
PUT         http://[hostname]/todo/api/v1.0/tasks/[task_id]  更新任务
DELETE      http://[hostname]/todo/api/v1.0/tasks/[task_id]  删除任务
==========  ================================================ ============================
```

我们定义的任务有如下一些属性:

- **id**: 任务的唯一标识符。数字类型。
- **title**: 简短的任务描述。字符串类型。
- **description**: 具体的任务描述。文本类型。
- **done**: 任务完成的状态。布尔值。

目前为止关于我们的 web service 的设计基本完成。剩下的事情就是实现它！

# Flask 框架的简介

如果你读过 Flask Mega-Tutorial 系列，就会知道 Flask 是一个简单却十分强大的 Python web 框架。

在我们深入研究 web services 的细节之前，让我们回顾一下一个普通的 Flask Web 应用程序的结构。

我会首先假设你知道 Python 在你的平台上工作的基本知识。 我将讲解的例子是工作在一个类 Unix 操作系统。简而言之，这意味着它们能工作在 Linux，Mac OS X 和 Windows(如果你使用Cygwin)。 如果你使用 Windows 上原生的 Python 版本的话，命令会有所不同。

让我们开始在一个虚拟环境上安装 Flask。如果你的系统上没有 virtualenv，你可以从 https://pypi.python.org/pypi/virtualenv 上下载:

```
$ mkdir todo-api
$ cd todo-api
$ virtualenv flask
New python executable in flask/bin/python
Installing setuptools............................done.
Installing pip...................done.
$ flask/bin/pip install flask
```

既然已经安装了 Flask，现在开始创建一个简单地网页应用，我们把它放在一个叫 app.py 的文件中:

```
#!flask/bin/python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello, World!"

if __name__ == '__main__':
    app.run(debug=True)
```

为了运行这个程序我们必须执行 app.py:

```
$ chmod a+x app.py
$ ./app.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

现在你可以启动你的网页浏览器，输入 http://localhost:5000 看看这个小应用程序的效果。

简单吧？现在我们将这个应用程序转换成我们的 RESTful service！

# 使用 Python 和 Flask 实现 RESTful services

使用 Flask 构建 web services 是十分简单地，比我在 Mega-Tutorial 中构建的完整的服务端的应用程序要简单地多。

在 Flask 中有许多扩展来帮助我们构建 RESTful services，但是在我看来这个任务十分简单，没有必要使用 Flask 扩展。

我们 web service 的客户端需要添加、删除以及修改任务的服务，因此显然我们需要一种方式来存储任务。最直接的方式就是建立一个小型的数据库，但是数据库并不是本文的主体。学习在 Flask 中使用合适的数据库，我强烈建议阅读 Mega-Tutorial。

这里我们直接把任务列表存储在内存中，因此这些任务列表只会在 web 服务器运行中工作，在结束的时候就失效。 这种方式只是适用我们自己开发的 web 服务器，不适用于生产环境的 web 服务器， 这种情况一个合适的数据库的搭建是必须的。

我们现在来实现 web service 的第一个入口:

```
#!flask/bin/python
from flask import Flask, jsonify

app = Flask(__name__)

tasks = [
    {
        'id': 1,
        'title': u'Buy groceries',
        'description': u'Milk, Cheese, Pizza, Fruit, Tylenol',
        'done': False
    },
    {
        'id': 2,
        'title': u'Learn Python',
        'description': u'Need to find a good Python tutorial on the web',
        'done': False
    }
]

@app.route('/todo/api/v1.0/tasks', methods=['GET'])
def get_tasks():
    return jsonify({'tasks': tasks})

if __name__ == '__main__':
    app.run(debug=True)
```

正如你所见，没有多大的变化。我们创建一个任务的内存数据库，这里无非就是一个字典和数组。数组中的每一个元素都具有上述定义的任务的属性。

取代了首页，我们现在拥有一个 get_tasks 的函数，访问的 URI 为 /todo/api/v1.0/tasks，并且只允许 GET 的 HTTP 方法。

这个函数的响应不是文本，我们使用 JSON 数据格式来响应，Flask 的 jsonify 函数从我们的数据结构中生成。

使用网页浏览器来测试我们的 web service 不是一个最好的注意，因为网页浏览器上不能轻易地模拟所有的 HTTP 请求的方法。相反，我们会使用 curl。如果你还没有安装 curl 的话，请立即安装它。

通过执行 app.py，启动 web service。接着打开一个新的控制台窗口，运行以下命令:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 294
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 04:53:53 GMT

{
  "tasks": [
    {
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "done": false,
      "id": 1,
      "title": "Buy groceries"
    },
    {
      "description": "Need to find a good Python tutorial on the web",
      "done": false,
      "id": 2,
      "title": "Learn Python"
    }
  ]
}
```

我们已经成功地调用我们的 RESTful service 的一个函数！

现在我们开始编写 GET 方法请求我们的任务资源的第二个版本。这是一个用来返回单独一个任务的函数:

```
from flask import abort

@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods=['GET'])
def get_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    return jsonify({'task': task[0]})
```

第二个函数有些意思。这里我们得到了 URL 中任务的 id，接着 Flask 把它转换成 函数中的 task_id 的参数。

我们用这个参数来搜索我们的任务数组。如果我们的数据库中不存在搜索的 id，我们将会返回一个类似 404 的错误，根据 HTTP 规范的意思是 “资源未找到”。

如果我们找到相应的任务，那么我们只需将它用 jsonify 打包成 JSON 格式并将其发送作为响应，就像我们以前那样处理整个任务集合。

调用 curl 请求的结果如下:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 151
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:21:50 GMT

{
  "task": {
    "description": "Need to find a good Python tutorial on the web",
    "done": false,
    "id": 2,
    "title": "Learn Python"
  }
}
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/3
HTTP/1.0 404 NOT FOUND
Content-Type: text/html
Content-Length: 238
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:21:52 GMT

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server.</p><p>If you     entered the URL manual
```

当我们请求 id #2 的资源时候，我们获取到了，但是当我们请求 #3 的时候返回了 404 错误。有关错误奇怪的是返回的是 HTML 信息而不是 JSON，这是因为 Flask 按照默认方式生成 404 响应。由于这是一个 Web service 客户端希望我们总是以 JSON 格式回应，所以我们需要改善我们的 404 错误处理程序:

```
from flask import make_response

@app.errorhandler(404)
def not_found(error):
    return make_response(jsonify({'error': 'Not found'}), 404)
```

我们会得到一个友好的错误提示:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/3
HTTP/1.0 404 NOT FOUND
Content-Type: application/json
Content-Length: 26
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:36:54 GMT

{
  "error": "Not found"
}
```

接下来就是 POST 方法，我们用来在我们的任务数据库中插入一个新的任务:

```
from flask import request

@app.route('/todo/api/v1.0/tasks', methods=['POST'])
def create_task():
    if not request.json or not 'title' in request.json:
        abort(400)
    task = {
        'id': tasks[-1]['id'] + 1,
        'title': request.json['title'],
        'description': request.json.get('description', ""),
        'done': False
    }
    tasks.append(task)
    return jsonify({'task': task}), 201
```

添加一个新的任务也是相当容易地。只有当请求以 JSON 格式形式，request.json 才会有请求的数据。如果没有数据，或者存在数据但是缺少 title 项，我们将会返回 400，这是表示请求无效。

接着我们会创建一个新的任务字典，使用最后一个任务的 id + 1 作为该任务的 id。我们允许 description 字段缺失，并且假设 done 字段设置成 False。

我们把新的任务添加到我们的任务数组中，并且把新添加的任务和状态 201 响应给客户端。

使用如下的 curl 命令来测试这个新的函数:

```
$ curl -i -H "Content-Type: application/json" -X POST -d '{"title":"Read a book"}' http:/
HTTP/1.0 201 Created
Content-Type: application/json
Content-Length: 104
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:56:21 GMT

{
  "task": {
    "description": "",
    "done": false,
    "id": 3,
    "title": "Read a book"
  }
}
```

注意：如果你在 Windows 上并且运行 Cygwin 版本的 curl，上面的命令不会有任何问题。然而，如果你使用原生的 curl，命令会有些不同:

```
curl -i -H "Content-Type: application/json" -X POST -d "{"""title""":"""Read a book"""}"
```

当然在完成这个请求后，我们可以得到任务的更新列表:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 423
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:57:44 GMT

{
  "tasks": [
    {
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "done": false,
      "id": 1,
      "title": "Buy groceries"
    },
    {
      "description": "Need to find a good Python tutorial on the web",
      "done": false,
      "id": 2,
      "title": "Learn Python"
    },
    {
      "description": "",
      "done": false,
      "id": 3,
      "title": "Read a book"
    }
  ]
}
```

剩下的两个函数如下所示:

```
@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods=['PUT'])
def update_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    if not request.json:
        abort(400)
    if 'title' in request.json and type(request.json['title']) != unicode:
        abort(400)
    if 'description' in request.json and type(request.json['description']) is not unicode
        abort(400)
    if 'done' in request.json and type(request.json['done']) is not bool:
        abort(400)
    task[0]['title'] = request.json.get('title', task[0]['title'])
    task[0]['description'] = request.json.get('description', task[0]['description'])
    task[0]['done'] = request.json.get('done', task[0]['done'])
    return jsonify({'task': task[0]})

@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods=['DELETE'])
def delete_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    tasks.remove(task[0])
    return jsonify({'result': True})
```

delete_task 函数没有什么特别的。对于 update_task 函数，我们需要严格地检查输入的参数以防止可能的问题。我们需要确保在我们把它更新到数据库之前，任何客户端提供我们的是预期的格式。

更新任务 #2 的函数调用如下所示:

```
$ curl -i -H "Content-Type: application/json" -X PUT -d '{"done":true}' http://localhost:
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 170
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 07:10:16 GMT

{
  "task": [
    {
      "description": "Need to find a good Python tutorial on the web",
      "done": true,
      "id": 2,
      "title": "Learn Python"
    }
  ]
}
```

# 优化 web service 接口

目前 API 的设计的问题就是迫使客户端在任务标识返回后去构造 URIs。这对于服务器是十分简单的，但是间接地迫使客户端知道这些 URIs 是如何构造的，这将会阻碍我们以后变更这些 URIs。

不直接返回任务的 ids，我们直接返回控制这些任务的完整的 URI，以便客户端可以随时使用这些 URIs。为此，我们可以写一个小的辅助函数生成一个 “公共” 版本任务发送到客户端:

```
from flask import url_for

def make_public_task(task):
    new_task = {}
    for field in task:
        if field == 'id':
            new_task['uri'] = url_for('get_task', task_id=task['id'], _external=True)
        else:
            new_task[field] = task[field]
    return new_task
```

这里所有做的事情就是从我们数据库中取出任务并且创建一个新的任务，这个任务的 id 字段被替换成通过 Flask 的 url_for 生成的 uri 字段。

当我们返回所有的任务列表的时候，在发送到客户端之前通过这个函数进行处理:

```
@app.route('/todo/api/v1.0/tasks', methods=['GET'])
def get_tasks():
    return jsonify({'tasks': map(make_public_task, tasks)})
```

这里就是客户端获取任务列表的时候得到的数据:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 406
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 18:16:28 GMT

{
  "tasks": [
    {
      "title": "Buy groceries",
      "done": false,
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/1"
    },
    {
      "title": "Learn Python",
      "done": false,
      "description": "Need to find a good Python tutorial on the web",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/2"
    }
  ]
}
```

我们将会把上述的方式应用到其它所有的函数上以确保客户端一直看到 URIs 而不是 ids。

# 加强 RESTful web service 的安全性

我们已经完成了我们 web service 的大部分功能，但是仍然有一个问题。我们的 web service 对任何人都是公开的，这并不是一个好主意。

我们有一个可以管理我们的待办事项完整的 web service，但在当前状态下的 web service 是开放给所有的客户端。 如果一个陌生人弄清我们的 API 是如何工作的，他或她可以编写一个客户端访问我们的 web service 并且毁坏我们的数据。

大部分初级的教程会忽略这个问题并且到此为止。在我看来这是一个很严重的问题，我必须指出。

确保我们的 web service 安全服务的最简单的方法是要求客户端提供一个用户名和密码。在常规的 web 应用程序会提供一个登录的表单用来认证，并且服务器会创建一个会话为登录的用户以后的操作使用，会话的 id 以 cookie 形式存储在客户端浏览器中。然而 REST 的规则之一就是 “无状态”， 因此我们必须要求客户端在每一次请求中提供认证的信息。

我们一直试着尽可能地坚持 HTTP 标准协议。既然我们需要实现认证我们需要在 HTTP 上下文中去完成，HTTP 协议提供了两种认证机制: Basic 和 Digest。

有一个小的 Flask 扩展能够帮助我们，我们可以先安装 Flask-HTTPAuth:

```
$ flask/bin/pip install flask-httpauth
```

比方说，我们希望我们的 web service 只让访问用户名 miguel 和密码 python 的客户端访问。我们可以设置一个基本的 HTTP 验证如下:

```
from flask.ext.httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()

@auth.get_password
def get_password(username):
    if username == 'miguel':
        return 'python'
    return None

@auth.error_handler
def unauthorized():
    return make_response(jsonify({'error': 'Unauthorized access'}), 401)
```

get_password 函数是一个回调函数，Flask-HTTPAuth 使用它来获取给定用户的密码。在一个更复杂的系统中，这个函数是需要检查一个用户数据库，但是在我们的例子中只有单一的用户因此没有必要。

error_handler 回调函数是用于给客户端发送未授权错误代码。像我们处理其它的错误代码，这里我们定制一个包含 JSON 数据格式而不是 HTML 的响应。

随着认证系统的建立，所剩下的就是把需要认证的函数添加 @auth.login_required 装饰器。例如:

```
@app.route('/todo/api/v1.0/tasks', methods=['GET'])
@auth.login_required
def get_tasks():
    return jsonify({'tasks': tasks})
```

如果现在要尝试使用 curl 调用这个函数我们会得到:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 36
WWW-Authenticate: Basic realm="Authentication Required"
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 06:41:14 GMT

{
  "error": "Unauthorized access"
}
```

为了能够调用这个函数我们必须发送我们的认证凭据:

```
$ curl -u miguel:python -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 316
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 06:46:45 GMT

{
  "tasks": [
    {
      "title": "Buy groceries",
      "done": false,
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/1"
    },
    {
      "title": "Learn Python",
      "done": false,
      "description": "Need to find a good Python tutorial on the web",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/2"
    }
  ]
}
```

认证扩展给予我们很大的自由选择哪些函数需要保护，哪些函数需要公开。

为了确保登录信息的安全应该使用 HTTP 安全服务器(例如: https://...)，这样客户端和服务器之间的通信都是加密的，以防止传输过程中第三方看到认证的凭据。

让人不舒服的是当请求收到一个 401 的错误，网页浏览都会跳出一个丑陋的登录框，即使请求是在后台发生的。因此如果我们要实现一个完美的 web 服务器的话，我们就需要禁止跳转到浏览器显示身份验证对话框，让我们的客户端应用程序自己处理登录。

一个简单的方式就是不返回 401 错误。403 错误是一个令人青睐的替代，403 错误表示 “禁止” 的错误:

```
@auth.error_handler
def unauthorized():
    return make_response(jsonify({'error': 'Unauthorized access'}), 403)
```

# 可能的改进

我们编写的小型的 web service 还可以在不少的方面进行改进。

对于初学者来说，一个真正的 web service 需要一个真实的数据库进行支撑。我们现在使用的内存数据结构会有很多限制不应该被用于真正的应用。

另外一个可以提高的领域就是处理多用户。如果系统支持多用户的话，不同的客户端可以发送不同的认证凭证获取相应用户的任务列表。在这样一个系统中的话，我们需要第二个资源就是用户。在用户资源上的 POST 的请求代表注册换一个新用户。一个 GET 请求表示客户端获取一个用户的信息。一个 PUT 请求表示更新用户信息，比如可能是更新邮箱地址。一个 DELETE 请求表示删除用户账号。

GET 检索任务列表请求可以在几个方面进行扩展。首先可以携带一个可选的页的参数，以便客户端请求任务的一部分。另外，这种扩展更加有用：允许按照一定的标准筛选。比如，用户只想要看到完成的任务，或者只想看到任务的标题以 A 字母开头。所有的这些都可以作为 URL 的一个参数项。

# 使用 Flask-RESTful 设计 RESTful API

前面我已经用 Flask 实现了一个 RESTful 服务器。今天我们将会使用 Flask-RESTful 来实现同一个 RESTful 服务器，Flask-RESTful 是一个可以简化 APIs 的构建的 Flask 扩展。

## RESTful 服务器

作为一个提醒， 这里就是待完成事项列表 web service 所提供的方法的定义:

```
==========  ================================================  =============================
HTTP 方法   URL                                               动作
==========  ================================================  =============================
GET         http://[hostname]/todo/api/v1.0/tasks             检索任务列表
GET         http://[hostname]/todo/api/v1.0/tasks/[task_id]   检索某个任务
POST        http://[hostname]/todo/api/v1.0/tasks             创建新任务
PUT         http://[hostname]/todo/api/v1.0/tasks/[task_id]   更新任务
DELETE      http://[hostname]/todo/api/v1.0/tasks/[task_id]   删除任务
==========  ================================================ =============================
```

这个服务唯一的资源叫做“任务”，它有如下一些属性:

- **id**: 任务的唯一标识符。数字类型。
- **title**: 简短的任务描述。字符串类型。
- **description**: 具体的任务描述。文本类型。
- **done**: 任务完成的状态。布尔值。

## 路由

在上一遍文章中，我使用了 Flask 的视图函数来定义所有的路由。

Flask-RESTful 提供了一个 Resource 基础类，它能够定义一个给定 URL 的一个或者多个 HTTP 方法。例如，定义一个可以使用 HTTP 的 GET, PUT 以及 DELETE 方法的 User 资源，你的代码可以如下:

```
from flask import Flask
from flask.ext.restful import Api, Resource

app = Flask(__name__)
api = Api(app)

class UserAPI(Resource):
    def get(self, id):
        pass

    def put(self, id):
        pass

    def delete(self, id):
        pass

api.add_resource(UserAPI, '/users/<int:id>', endpoint = 'user')
```

add_resource 函数使用指定的 endpoint 注册路由到框架上。如果没有指定 endpoint，Flask-RESTful 会根据类名生成一个，但是有时候有些函数比如 url_for 需要 endpoint，因此我会明确给 endpoint 赋值。

我的待办事项 API 定义两个 URLs：/todo/api/v1.0/tasks（获取所有任务列表），以及 /todo/api/v1.0/tasks/（获取单个任务）。我们现在需要两个资源:

```
class TaskListAPI(Resource):
    def get(self):
        pass

    def post(self):
        pass

class TaskAPI(Resource):
    def get(self, id):
        pass

    def put(self, id):
        pass

    def delete(self, id):
        pass

api.add_resource(TaskListAPI, '/todo/api/v1.0/tasks', endpoint = 'tasks')
api.add_resource(TaskAPI, '/todo/api/v1.0/tasks/<int:id>', endpoint = 'task')
```

# 解析以及验证请求

当我在以前的文章中实现此服务器的时候，我自己对请求的数据进行验证。例如，在之前版本中如何处理 PUT 的:

```
@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods = ['PUT'])
@auth.login_required
def update_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    if not request.json:
        abort(400)
    if 'title' in request.json and type(request.json['title']) != unicode:
        abort(400)
    if 'description' in request.json and type(request.json['description']) is not unicode
        abort(400)
    if 'done' in request.json and type(request.json['done']) is not bool:
        abort(400)
    task[0]['title'] = request.json.get('title', task[0]['title'])
    task[0]['description'] = request.json.get('description', task[0]['description'])
    task[0]['done'] = request.json.get('done', task[0]['done'])
    return jsonify( { 'task': make_public_task(task[0]) } )
```

在这里, 我必须确保请求中给出的数据在使用之前是有效，这样使得函数变得又臭又长。

Flask-RESTful 提供了一个更好的方式来处理数据验证，它叫做 RequestParser 类。这个类工作方式类似命令行解析工具 argparse。

首先，对于每一个资源需要定义参数以及怎样验证它们:

```
from flask.ext.restful import reqparse

class TaskListAPI(Resource):
    def __init__(self):
        self.reqparse = reqparse.RequestParser()
        self.reqparse.add_argument('title', type = str, required = True,
            help = 'No task title provided', location = 'json')
        self.reqparse.add_argument('description', type = str, default = "", location = 'j
        super(TaskListAPI, self).__init__()

    # ...

class TaskAPI(Resource):
    def __init__(self):
        self.reqparse = reqparse.RequestParser()
        self.reqparse.add_argument('title', type = str, location = 'json')
        self.reqparse.add_argument('description', type = str, location = 'json')
        self.reqparse.add_argument('done', type = bool, location = 'json')
        super(TaskAPI, self).__init__()

    # ...
```

在 TaskListAPI 资源中，POST 方法是唯一接收参数的。参数“标题”是必须的，因此我定义一个缺少“标题”的错误信息。当客户端缺少这个参数的时候，Flask-RESTful 将会把这个错误信息作为响应发送给客户端。“描述”字段是可选的，当缺少这个字段的时候，默认的空字符串将会被使用。一个有趣的方面就是 RequestParser 类默认情况下在 request.values 中查找参数，因此 location 可选参数必须被设置以表明请求过来的参数是 request.json 格式的。

TaskAPI 资源的参数处理是同样的方式，但是有少许不同。PUT 方法需要解析参数，并且这个方法的所有参数都是可选的。

当请求解析器被初始化，解析和验证一个请求是很容易的。 例如，请注意 TaskAPI.put() 方法变的多么地简单:

```
def put(self, id):
    task = filter(lambda t: t['id'] == id, tasks)
    if len(task) == 0:
        abort(404)
    task = task[0]
    args = self.reqparse.parse_args()
    for k, v in args.iteritems():
        if v != None:
            task[k] = v
    return jsonify( { 'task': make_public_task(task) } )
```

使用 Flask-RESTful 来处理验证的另一个好处就是没有必要单独地处理类似 HTTP 400 错误，Flask-RESTful 会来处理这些。

# 生成响应

原来设计的 REST 服务器使用 Flask 的 jsonify 函数来生成响应。Flask-RESTful 会自动地处理转换成 JSON 数据格式，因此下面的代码需要替换:

```
return jsonify( { 'task': make_public_task(task) } )
```

现在需要写成这样:

```
return { 'task': make_public_task(task) }
```

Flask-RESTful 也支持自定义状态码，如果有必要的话:

```
return { 'task': make_public_task(task) }, 201
```

Flask-RESTful 还有更多的功能。make_public_task 能够把来自原始服务器上的任务从内部形式包装成客户端想要的外部形式。最典型的就是把任务的 id 转成 uri。Flask-RESTful 就提供一个辅助函数能够很优雅地做到这样的转换，不仅仅能够把 id 转成 uri 并且能够转换其他的参数:

```
from flask.ext.restful import fields, marshal

task_fields = {
    'title': fields.String,
    'description': fields.String,
    'done': fields.Boolean,
    'uri': fields.Url('task')
}

class TaskAPI(Resource):
    # ...

    def put(self, id):
        # ...
        return { 'task': marshal(task, task_fields) }
```

task_fields 结构用于作为 marshal 函数的模板。fields.Uri 是一个用于生成一个 URL 的特定的参数。 它需要的参数是 endpoint。

## 认证

在 REST 服务器中的路由都是由 HTTP 基本身份验证保护着。在最初的那个服务器是通过使用 Flask-HTTPAuth 扩展来实现的。

因为 Resouce 类是继承自 Flask 的 MethodView，它能够通过定义 decorators 变量并且把装饰器赋予给它:

```
from flask.ext.httpauth import HTTPBasicAuth
# ...
auth = HTTPBasicAuth()
# ...

class TaskAPI(Resource):
    decorators = [auth.login_required]
    # ...

class TaskAPI(Resource):
    decorators = [auth.login_required]
    # ...
```

# 使用 Flask 设计 RESTful 的认证

今天我将要展示一个简单，不过很安全的方式用来保护使用 Flask 编写的 API，它是使用密码或者令牌认证的。

## 示例代码

本文使用的代码能够在 github 上找到: REST-auth 。

# 用户数据库

为了让给出的示例看起来像真实的项目，这里我将使用 Flask-SQLAlchemy 来构建用户数据库模型并且存储到数据库中。

用户的数据库模型是十分简单的。对于每一个用户，username 和 password_hash 将会被存储:

```
class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key = True)
    username = db.Column(db.String(32), index = True)
    password_hash = db.Column(db.String(128))
```

出于安全原因，用户的原始密码将不被存储，密码在注册时被散列后存储到数据库中。使用散列密码的话，如果用户数据库不小心落入恶意攻击者的手里，他们也很难从散列中解析到真实的密码。

密码 决不能 很明确地存储在用户数据库中。

# 密码散列

为了创建密码散列，我将会使用 PassLib 库，一个专门用于密码散列的 Python 包。

PassLib 提供了多种散列算法供选择。custom_app_context 是一个易于使用的基于 sha256_crypt 的散列算法。

User 用户模型需要增加两个新方法来增加密码散列和密码验证功能:

```
from passlib.apps import custom_app_context as pwd_context

    class User(db.Model):
        # ...

        def hash_password(self, password):
            self.password_hash = pwd_context.encrypt(password)

        def verify_password(self, password):
            return pwd_context.verify(password, self.password_hash)
```

hash_password() 函数接受一个明文的密码作为参数并且存储明文密码的散列。当一个新用户注册到服务器或者当用户修改密码的时候，这个函数将被调用。

verify_password() 函数接受一个明文的密码作为参数并且当密码正确的话返回 True 或者密码错误的话返回 False。这个函数当用户提供和需要验证凭证的时候调用。

你可能会问如果原始密码散列后如何验证原始密码的？

散列算法是单向函数，这就是意味着它们能够用于根据密码生成散列，但是无法根据生成的散列逆向猜测出原密码。然而这些算法是具有确定性的，给定相同的输入它们总会得到相同的输出。PassLib 所有需要做的就是验证密码，通过使用注册时候同一个函数散列密码并且同存储在数据库中的散列值进行比较。

# 用户注册

在本文例子中，一个客户端可以使用 POST 请求到 /api/users 上注册一个新用户。请求的主体必须是一个包含 username 和 password 的 JSON 格式的对象。

Flask 中的路由的实现如下所示:

```
@app.route('/api/users', methods = ['POST'])
def new_user():
    username = request.json.get('username')
    password = request.json.get('password')
    if username is None or password is None:
        abort(400) # missing arguments
    if User.query.filter_by(username = username).first() is not None:
        abort(400) # existing user
    user = User(username = username)
    user.hash_password(password)
    db.session.add(user)
    db.session.commit()
    return jsonify({ 'username': user.username }), 201, {'Location': url_for('get_user',
```

这个函数是十分简单地。参数 username 和 password 是从请求中携带的 JSON 数据中获取，接着验证它们。

如果参数通过验证的话，新的 User 实例被创建。username 赋予给 User，接着使用 hash_password 方法散列密码。用户最终被写入数据库中。

响应的主体是一个表示用户的 JSON 对象，201 状态码以及一个指向新创建的用户的 URI 的 HTTP 头信息：Location。

注意：get_user 函数可以在 github 上找到完整的代码。

这里是一个用户注册的请求，发送自 curl:

```
$ curl -i -X POST -H "Content-Type: application/json" -d '{"username":"miguel","password"
HTTP/1.0 201 CREATED
Content-Type: application/json
Content-Length: 27
Location: http://127.0.0.1:5000/api/users/1
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Thu, 28 Nov 2013 19:56:39 GMT

{
  "username": "miguel"
}
```

需要注意地是在真实的应用中这里可能会使用安全的的 HTTP (譬如：HTTPS)。如果用户登录的凭证是通过明文在网络传输的话，任何对 API 的保护措施是毫无意义的。

# 基于密码的认证

现在我们假设存在一个资源通过一个 API 暴露给那些必须注册的用户。这个资源是通过 URL: /api/resource 能够访问到。

为了保护这个资源，我们将使用 HTTP 基本身份认证，但是不是自己编写完整的代码来实现它，而是让 Flask-HTTPAuth 扩展来为我们做。

使用 Flask-HTTPAuth，通过添加 login_required 装饰器可以要求相应的路由必须进行认证:

```
from flask.ext.httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()

@app.route('/api/resource')
@auth.login_required
def get_resource():
    return jsonify({ 'data': 'Hello, %s!' % g.user.username })
```

但是，Flask-HTTPAuth 需要给予更多的信息来验证用户的认证，当然 Flask-HTTPAuth 有着许多的选项，它取决于应用程序实现的安全级别。

能够提供最大自由度的选择(可能这也是唯一兼容 PassLib 散列)就是选用 verify_password 回调函数，这个回调函数将会根据提供的 username 和 password 的组合的，返回 True(通过验证) 或者 Flase(未通过验证)。Flask-HTTPAuth 将会在需要验证 username 和 password 对的时候调用这个回调函数。

verify_password 回调函数的实现如下:

```
@auth.verify_password
def verify_password(username, password):
    user = User.query.filter_by(username = username).first()
    if not user or not user.verify_password(password):
        return False
    g.user = user
    return True
```

这个函数将会根据 username 找到用户，并且使用 verify_password() 方法验证密码。如果认证通过的话，用户对象将会被存储在 Flask 的 g 对象中，这样视图就能使用它。

这里是用 curl 请求只允许注册用户获取的保护资源:

```
$ curl -u miguel:python -i -X GET http://127.0.0.1:5000/api/resource
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 30
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Thu, 28 Nov 2013 20:02:25 GMT

{
  "data": "Hello, miguel!"
}
```

如果登录失败的话，会得到下面的内容:

```
$ curl -u miguel:ruby -i -X GET http://127.0.0.1:5000/api/resource
HTTP/1.0 401 UNAUTHORIZED
Content-Type: text/html; charset=utf-8
Content-Length: 19
WWW-Authenticate: Basic realm="Authentication Required"
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Thu, 28 Nov 2013 20:03:18 GMT

Unauthorized Access
```

这里我再次重申在实际的应用中，请使用安全的 HTTP。

# 基于令牌的认证

每次请求必须发送 username 和 password 是十分不方便，即使是通过安全的 HTTP 传输的话还是存在风险，因为客户端必须要存储不加密的认证凭证，这样才能在每次请求中发送。

一种基于之前解决方案的优化就是使用令牌来验证请求。

我们的想法是客户端应用程序使用认证凭证交换了认证令牌，接下来的请求只发送认证令牌。

令牌是具有有效时间，过了有效时间后，令牌变成无效，需要重新获取新的令牌。令牌的潜在风险在于生成令牌的算法比较弱，但是有效期较短可以减少风险。

有很多的方法可以加强令牌。一个简单的强化方式就是根据存储在数据库中的用户以及密码生成一个随机的特定长度的字符串，可能过期日期也在里面。令牌就变成了明文密码的重排，这样就能很容易地进行字符串对比，还能对过期日期进行检查。

更加完善的实现就是不需要服务器端进行任何存储操作，使用加密的签名作为令牌。这种方式有很多的优点，能够根据用户信息生成相关的签名，并且很难被篡改。

Flask 使用类似的方式处理 cookies 的。这个实现依赖于一个叫做 itsdangerous 的库，我们这里也会采用它。

令牌的生成以及验证将会被添加到 User 模型中，其具体实现如下:

```
from itsdangerous import TimedJSONWebSignatureSerializer as Serializer

class User(db.Model):
    # ...

    def generate_auth_token(self, expiration = 600):
        s = Serializer(app.config['SECRET_KEY'], expires_in = expiration)
        return s.dumps({ 'id': self.id })

    @staticmethod
    def verify_auth_token(token):
        s = Serializer(app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except SignatureExpired:
            return None # valid token, but expired
        except BadSignature:
            return None # invalid token
        user = User.query.get(data['id'])
        return user
```

generate_auth_token() 方法生成一个以用户 id 值为值，’id’ 为关键字的字典的加密令牌。令牌中同时加入了一个过期时间，默认为十分钟(600 秒)。

验证令牌是在 verify_auth_token() 静态方法中实现的。静态方法被使用在这里，是因为一旦令牌被解码了用户才可得知。如果令牌被解码了，相应的用户将会被查询出来并且返回。

API 需要一个获取令牌的新函数，这样客户端才能申请到令牌:

```
@app.route('/api/token')
@auth.login_required
def get_auth_token():
    token = g.user.generate_auth_token()
    return jsonify({ 'token': token.decode('ascii') })
```

注意：这个函数是使用了 auth.login_required 装饰器，也就是说需要提供 username 和 password。

剩下来的就是决策客户端怎样在请求中包含这个令牌。

HTTP 基本认证方式不特别要求 usernames 和 passwords 用于认证，在 HTTP 头中这两个字段可以用于任何类型的认证信息。基于令牌的认证，令牌可以作为 username 字段，password 字段可以忽略。

这就意味着服务器需要同时处理 username 和 password 作为认证，以及令牌作为 username 的认证方式。verify_password 回调函数需要同时支持这两种方式:

```
@auth.verify_password
def verify_password(username_or_token, password):
    # first try to authenticate by token
    user = User.verify_auth_token(username_or_token)
    if not user:
        # try to authenticate with username/password
        user = User.query.filter_by(username = username_or_token).first()
        if not user or not user.verify_password(password):
            return False
    g.user = user
    return True
```

新版的 verify_password 回调函数会尝试认证两次。首先它会把 username 参数作为令牌进行认证。如果没有验证通过的话，就会像基于密码认证的一样，验证 username 和 password。

如下的 curl 请求能够获取一个认证的令牌:

```
$ curl -u miguel:python -i -X GET http://127.0.0.1:5000/api/token
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 139
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Thu, 28 Nov 2013 20:04:15 GMT

{
  "token": "eyJhbGciOiJIUzI1NiIsImV4cCI6MTM4NTY2OTY1NSwiaWF0IjoxMzg1NjY5MDU1fQ.eyJpZCI6MX
}
```

现在可以使用令牌获取资源:

```
$ curl -u eyJhbGciOiJIUzI1NiIsImV4cCI6MTM4NTY2OTY1NSwiaWF0IjoxMzg1NjY5MDU1fQ.eyJpZCI6MX0.
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 30
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Thu, 28 Nov 2013 20:05:08 GMT

{
  "data": "Hello, miguel!"
}
```

需要注意的是这里并没有使用密码。

# OAuth 认证

当我们讨论 RESTful 认证的时候，OAuth 协议经常被提及到。

那么什么是 OAuth？

OAuth 可以有很多的含义。最通常就是一个应用程序允许其它应用程序的用户的接入或者使用服务，但是用户必须使用应用程序提供的登录凭证。我建议阅读者可以浏览 OAuth 了解更多知识。

# Flask-SQLAlchemy

Flask-SQLAlchemy 是一个为您的 Flask 应用增加 SQLAlchemy 支持的扩展。它需要 SQLAlchemy 0.6 或者更高的版本。它致力于简化在 Flask 中 SQLAlchemy 的使用，提供了有用的默认值和额外的助手来更简单地完成常见任务。

# 快速入门

Flask-SQLAlchemy 使用起来非常有趣，对于基本应用十分容易使用，并且对于大型项目易于扩展。有关完整的指南，请参阅 `SQLAlchemy` 的 API 文档。

## 一个最小应用

常见情况下对于只有一个 Flask 应用，所有您需要做的事情就是创建 Flask 应用，选择加载配置接着创建 `SQLAlchemy` 对象时候把 Flask 应用传递给它作为参数。

一旦创建，这个对象就包含 `sqlalchemy` 和 `sqlalchemy.orm` 中的所有函数和助手。此外它还提供一个名为 `Model` 的类，用于作为声明模型时的 delarative 基类：

```
from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    email = db.Column(db.String(120), unique=True)

    def __init__(self, username, email):
        self.username = username
        self.email = email

    def __repr__(self):
        return '<User %r>' % self.username
```

为了创建初始数据库，只需要从交互式 Python shell 中导入 `db` 对象并且调用 `SQLAlchemy.create_all()` 方法来创建表和数据库：

```
>>> from yourapplication import db
>>> db.create_all()
```

Boom, 您的数据库已经生成。现在来创建一些用户：

```
>>> from yourapplication import User
>>> admin = User('admin', 'admin@example.com')
>>> guest = User('guest', 'guest@example.com')
```

但是它们还没有真正地写入到数据库中，因此让我们来确保它们已经写入到数据库中:

```
>>> db.session.add(admin)
>>> db.session.add(guest)
>>> db.session.commit()
```

访问数据库中的数据也是十分简单的:

```
>>> users = User.query.all()
[<User u'admin'>, <User u'guest'>]
>>> admin = User.query.filter_by(username='admin').first()
<User u'admin'>
```

# 简单的关系

SQLAlchemy 连接到关系型数据库，关系型数据最擅长的东西就是关系。因此，我们将创建一个使用两张相互关联的表的应用作为例子:

```
from datetime import datetime

class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80))
    body = db.Column(db.Text)
    pub_date = db.Column(db.DateTime)

    category_id = db.Column(db.Integer, db.ForeignKey('category.id'))
    category = db.relationship('Category',
        backref=db.backref('posts', lazy='dynamic'))

    def __init__(self, title, body, category, pub_date=None):
        self.title = title
        self.body = body
        if pub_date is None:
            pub_date = datetime.utcnow()
        self.pub_date = pub_date
        self.category = category

    def __repr__(self):
        return '<Post %r>' % self.title

class Category(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50))

    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return '<Category %r>' % self.name
```

首先让我们创建一些对象:

```
>>> py = Category('Python')
>>> p = Post('Hello Python!', 'Python is pretty cool', py)
>>> db.session.add(py)
>>> db.session.add(p)
```

现在因为我们在 backref 中声明了 `posts` 作为动态关系，查询显示为:

```
>>> py.posts
<sqlalchemy.orm.dynamic.AppenderBaseQuery object at 0x1027d37d0>
```

它的行为像一个普通的查询对象，因此我们可以查询与我们测试的 “Python” 分类相关的所有文章(posts):

```
>>> py.posts.all()
[<Post 'Hello Python!'>]
```

# 启蒙之路

您仅需要知道与普通的 SQLAlchemy 不同之处:

1. `SQLAlchemy` 允许您访问下面的东西:
    - `sqlalchemy` 和 `sqlalchemy.orm` 下所有的函数和类
    - 一个叫做 `session` 的预配置范围的会话(session)
    - `metadata` 属性
    - `engine` 属性
    - `SQLAlchemy.create_all()` 和 `SQLAlchemy.drop_all()`，根据模型用来创建以及删除表格的方法
    - 一个 `Model` 基类，即是一个已配置的声明(declarative)的基础(base)
2. `Model` 声明基类行为类似一个常规的 Python 类，不过有个 `query` 属性，可以用来查询模型 ( `Model` 和 `BaseQuery` )
3. 您必须提交会话，但是没有必要在每个请求后删除它(session)，Flask-SQLAlchemy 会帮您完成删除操作。

# 引入上下文

如果您计划只使用一个应用程序，您大可跳过这一章节。只需要把您的应用程序传给 `SQLAlchemy` 构造函数，一般情况下您就设置好了。然而您想要使用不止一个应用或者在一个函数中动态地创建应用的话，您需要仔细阅读。

如果您在一个函数中定义您的应用，但是 `SQLAlchemy` 对象是全局的，后者如何知道前者了？答案就是 `init_app()` 函数:

```
from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy

db = SQLAlchemy()

def create_app():
    app = Flask(__name__)
    db.init_app(app)
    return app
```

它所做的是准备应用以与 `SQLAlchemy` 共同工作。然而现在不把 `SQLAlchemy` 对象绑定到您的应用。为什么不这么做？ 因为那里可能创建不止一个应用。

那么 `SQLAlchemy` 是如何知道您的应用的？您必须配置一个应用上下文。如果您在一个 Flask 视图函数中进行工作，这会自动实现。但如果您在交互式的 shell 中，您需要手动这么做。(参阅 创建应用上下文 )。

简而言之，像这样做:

```
>>> from yourapp import create_app
>>> app = create_app()
>>> app.app_context().push()
```

在脚本里面使用 with 声明都样也有作用:

```
def my_function():
    with app.app_context():
        user = db.User(...)
        db.session.add(user)
        db.session.commit()
```

Flask-SQLAlchemy 里的一些函数也可以接受要在其上进行操作的应用作为参数:

```
>>> from yourapp import db, create_app
>>> db.create_all(app=create_app())
```

# 配置

下面是 Flask-SQLAlchemy 中存在的配置值。Flask-SQLAlchemy 从您的 Flask 主配置中加载这些值。 注意其中的一些在引擎创建后不能修改，所以确保尽早配置且不在运行时修改它们。

## 配置键

Flask-SQLAlchemy 扩展能够识别的配置键的清单:

| | |
|---|---|
| `SQLALCHEMY_DATABASE_URI` | 用于连接数据的数据库。例如：`sqlite:////tmp/test.db` `mysql://username:password@server/db` |
| `SQLALCHEMY_BINDS` | 一个映射绑定 (bind) 键到 SQLAlchemy 连接 URIs 的字典。 更多的信息请参阅 绑定多个数据库。 |
| `SQLALCHEMY_ECHO` | 如果设置成 `True` ，SQLAlchemy 将会记录所有 发到标准输出(stderr)的语句，这对调试很有帮助。 |
| `SQLALCHEMY_RECORD_QUERIES` | 可以用于显式地禁用或者启用查询记录。查询记录在调试或者测试模式下自动启用。更多信息请参阅 `get_debug_queries()` 。 |
| `SQLALCHEMY_NATIVE_UNICODE` | 可以用于显式地禁用支持原生的 unicode。这是 某些数据库适配器必须的（像在 Ubuntu 某些版本上的 PostgreSQL），当使用不合适的指定无编码的数据库 默认值时。 |
| `SQLALCHEMY_POOL_SIZE` | 数据库连接池的大小。默认是数据库引擎的默认值（通常是 5）。 |
| `SQLALCHEMY_POOL_TIMEOUT` | 指定数据库连接池的超时时间。默认是 10。 |
| `SQLALCHEMY_POOL_RECYCLE` | 自动回收连接的秒数。这对 MySQL 是必须的，默认情况下 MySQL 会自动移除闲置 8 小时或者以上的连接。 需要注意地是如果使用 MySQL 的话， Flask-SQLAlchemy 会自动地设置这个值为 2 小时。 |
| `SQLALCHEMY_MAX_OVERFLOW` | 控制在连接池达到最大值后可以创建的连接数。当这些额外的 连接回收到连接池后将会被断开和抛弃。 |
| `SQLALCHEMY_TRACK_MODIFICATIONS` | 如果设置成 `True` (默认情况)，Flask-SQLAlchemy 将会追踪对象的修改并且发送信号。这需要额外的内存， 如果不必要的可以禁用它。 |

New in version 0.8: 增加 `SQLALCHEMY_NATIVE_UNICODE` , `SQLALCHEMY_POOL_SIZE` , `SQLALCHEMY_POOL_TIMEOUT` 和 `SQLALCHEMY_POOL_RECYCLE` 配置键。

New in version 0.12: 增加 `SQLALCHEMY_BINDS` 配置键。

New in version 0.17: 增加 `SQLALCHEMY_MAX_OVERFLOW` 配置键。

New in version 2.0: 增加 `SQLALCHEMY_TRACK_MODIFICATIONS` 配置键。

# 连接 URI 格式

完整连接 URI 格式列表请跳转到 SQLAlchemy 下面的文档(支持的数据库)。这里展示了一些常见的连接字符串。

SQLAlchemy 把一个引擎的源表示为一个连同设定引擎选项的可选字符串参数的 URI。URI 的形式是:

```
dialect+driver://username:password@host:port/database
```

该字符串中的许多部分是可选的。如果没有指定驱动器，会选择默认的（确保在这种情况下不 包含 `+` ）。

Postgres:

```
postgresql://scott:tiger@localhost/mydatabase
```

MySQL:

```
mysql://scott:tiger@localhost/mydatabase
```

Oracle:

```
oracle://scott:tiger@127.0.0.1:1521/sidname
```

SQLite (注意开头的四个斜线):

```
sqlite:////absolute/path/to/foo.db
```

# 使用自定义的元数据和命名约定

你可以使用一个自定义的 `MetaData` 对象来构造 `SQLAlchemy` 对象。这允许你指定一个 自定义约束命名约定。这样做对数据库的迁移是很重要的。因为 SQL 没有定义一个标准的命名约定，无法保证数据库之间实现是兼容的。你可以自定义命名约定像 SQLAlchemy 文档建议那样:

```
from sqlalchemy import MetaData
from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy

convention = {
    "ix": 'ix_%(column_0_label)s',
    "uq": "uq_%(table_name)s_%(column_0_name)s",
    "ck": "ck_%(table_name)s_%(constraint_name)s",
    "fk": "fk_%(table_name)s_%(column_0_name)s_%(referred_table_name)s",
    "pk": "pk_%(table_name)s"
}

metadata = MetaData(naming_convention=convention)
db = SQLAlchemy(app, metadata=metadata)
```

更多关于 `MetaData` 的信息，请查看官方的文档。

# 声明模型

通常下，Flask-SQLAlchemy 的行为就像一个来自 `declarative` 扩展配置正确的 declarative 基类。因此，我们强烈建议您阅读 SQLAlchemy 文档以获取一个全面的参考。尽管如此，我们这里还是给出了最常用的示例。

需要牢记的事情:

- 您的所有模型的基类叫做 `db.Model`。它存储在您必须创建的 SQLAlchemy 实例上。 细节请参阅 快速入门。
- 有一些部分在 SQLAlchemy 上是必选的，但是在 Flask-SQLAlchemy 上是可选的。 比如表名是自动地为您设置好的，除非您想要覆盖它。它是从转成小写的类名派生出来的，即 “CamelCase” 转换为 “camel_case”。

## 简单示例

一个非常简单的例子:

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    email = db.Column(db.String(120), unique=True)

    def __init__(self, username, email):
        self.username = username
        self.email = email

    def __repr__(self):
        return '<User %r>' % self.username
```

用 `Column` 来定义一列。列名就是您赋值给那个变量的名称。如果您想要在表中使用不同的名称，您可以提供一个想要的列名的字符串作为可选第一个参数。主键用 `primary_key=True` 标记。可以把多个键标记为主键，此时它们作为复合主键。

列的类型是 `Column` 的第一个参数。您可以直接提供它们或进一步规定（比如提供一个长度）。下面的类型是最常用的:

| | |
|---|---|
| `Integer` | 一个整数 |
| `String` (size) | 有长度限制的字符串 |
| `Text` | 一些较长的 unicode 文本 |
| `DateTime` | 表示为 Python `datetime` 对象的 时间和日期 |
| `Float` | 存储浮点值 |
| `Boolean` | 存储布尔值 |
| `PickleType` | 存储为一个持久化的 Python 对象 |
| `LargeBinary` | 存储一个任意大的二进制数据 |

# 一对多(one-to-many)关系

最为常见的关系就是一对多的关系。因为关系在它们建立之前就已经声明，您可以使用 字符串来指代还没有创建的类(例如如果 `Person` 定义了一个到 `Article` 的关系，而 `Article` 在文件的后面才会声明)。

关系使用 `relationship()` 函数表示。然而外键必须用类 `sqlalchemy.schema.ForeignKey` 来单独声明:

```
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50))
    addresses = db.relationship('Address', backref='person',
                                lazy='dynamic')

class Address(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(50))
    person_id = db.Column(db.Integer, db.ForeignKey('person.id'))
```

`db.relationship()` 做了什么？这个函数返回一个可以做许多事情的新属性。在本案例中，我们让它指向 Address 类并加载多个地址。它如何知道会返回不止一个地址？因为 SQLALchemy 从您的声明中猜测了一个有用的默认值。 如果您想要一对一关系，您可以把 `uselist=False` 传给 `relationship()` 。

那么 `backref` 和 `lazy` 意味着什么了？`backref` 是一个在 `Address` 类上声明新属性的简单方法。您也可以使用 `my_address.person` 来获取使用该地址(address)的人(person)。 `lazy` 决定了 SQLAlchemy 什么时候从数据库中加载数据:

- `'select'` (默认值) 就是说 SQLAlchemy 会使用一个标准的 select 语句必要时一次加载数据。
- `'joined'` 告诉 SQLAlchemy 使用 `JOIN` 语句作为父级在同一查询中来加载关系。
- `'subquery'` 类似 `'joined'` ，但是 SQLAlchemy 会使用子查询。

- `'dynamic'` 在有多条数据的时候是特别有用的。不是直接加载这些数据，SQLAlchemy 会返回一个查询对象，在加载数据前您可以过滤（提取）它们。

您如何为反向引用（backrefs）定义惰性（lazy）状态？使用 `backref()` 函数:

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50))
    addresses = db.relationship('Address',
        backref=db.backref('person', lazy='joined'), lazy='dynamic')
```

## 多对多(many-to-many)关系

如果您想要用多对多关系，您需要定义一个用于关系的辅助表。对于这个辅助表， 强烈建议 不 使用模型，而是采用一个实际的表:

```
tags = db.Table('tags',
    db.Column('tag_id', db.Integer, db.ForeignKey('tag.id')),
    db.Column('page_id', db.Integer, db.ForeignKey('page.id'))
)

class Page(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    tags = db.relationship('Tag', secondary=tags,
        backref=db.backref('pages', lazy='dynamic'))

class Tag(db.Model):
    id = db.Column(db.Integer, primary_key=True)
```

这里我们配置 `Page.tags` 加载后作为标签的列表，因为我们并不期望每页出现太多的标签。而每个 tag 的页面列表（ `Tag.pages` ）是一个动态的反向引用。 正如上面提到的，这意味着您会得到一个可以发起 select 的查询对象。

# 选择(Select),插入(Insert), 删除(Delete)

现在您已经有了 *declared models*，是时候从数据库中查询数据。我们将会使用 快速入门 章节中定义的数据模型。

## 插入记录

在查询数据之前我们必须先插入数据。您的所有模型都应该有一个构造函数，如果您 忘记了，请确保加上一个。只有您自己使用这些构造函数而 SQLAlchemy 在内部不会使用它， 所以如何定义这些构造函数完全取决与您。

向数据库插入数据分为三个步骤:

1. 创建 Python 对象

2. 把它添加到会话
3. 提交会话

这里的会话不是 Flask 的会话，而是 Flask-SQLAlchemy 的会话。它本质上是一个 数据库事务的加强版本。它是这样工作的:

```
>>> from yourapp import User
>>> me = User('admin', 'admin@example.com')
>>> db.session.add(me)
>>> db.session.commit()
```

好吧，这不是很难吧。但是在您把对象添加到会话之前， SQLAlchemy 基本不考虑把它加到事务中。这是好事，因为您仍然可以放弃更改。比如想象 在一个页面上创建文章，但是您只想把文章传递给模板来预览渲染，而不是把它存进数据库。

调用 `add()` 函数会添加对象。它会发出一个 `INSERT` 语句给数据库，但是由于事务仍然没有提交，您不会立即得到返回的 ID 。如果您提交，您的用户会有一个 ID:

```
>>> me.id
1
```

## 删除记录

删除记录是十分类似的，使用 `delete()` 代替 `add()` :

```
>>> db.session.delete(me)
>>> db.session.commit()
```

## 查询记录

那么我们怎么从数据库中查询数据？为此，Flask-SQLAlchemy 在您的 `Model` 类上提供了 `query` 属性。当您访问它时，您会得到一个新的所有记录的查询对象。在使用 `all()` 或者 `first()` 发起查询之前可以使用方法 `filter()` 来过滤记录。如果您想要用主键查询的话，也可以使用 `get()` 。

下面的查询假设数据库中有如下条目:

| `id` | `username` | `email` |
|---|---|---|
| 1 | admin | admin@example.com |
| 2 | peter | peter@example.org |
| 3 | guest | guest@example.com |

通过用户名查询用户:

```
>>> peter = User.query.filter_by(username='peter').first()
>>> peter.id
1
>>> peter.email
u'peter@example.org'
```

同上但是查询一个不存在的用户名返回 `None`：

```
>>> missing = User.query.filter_by(username='missing').first()
>>> missing is None
True
```

使用更复杂的表达式查询一些用户:

```
>>> User.query.filter(User.email.endswith('@example.com')).all()
[<User u'admin'>, <User u'guest'>]
```

按某种规则对用户排序:

```
>>> User.query.order_by(User.username)
[<User u'admin'>, <User u'guest'>, <User u'peter'>]
```

限制返回用户的数量:

```
>>> User.query.limit(1).all()
[<User u'admin'>]
```

用主键查询用户:

```
>>> User.query.get(1)
<User u'admin'>
```

# 在视图中查询

当您编写 Flask 视图函数，对于不存在的条目返回一个 404 错误是非常方便的。因为这是一个很常见的问题，Flask-SQLAlchemy 为了解决这个问题提供了一个帮助函数。可以使用 `get_or_404()` 来代替 `get()`，使用 `first_or_404()` 来代替 `first()`。这样会抛出一个 404 错误，而不是返回 `None`：

```
@app.route('/user/<username>')
def show_user(username):
    user = User.query.filter_by(username=username).first_or_404()
    return render_template('show_user.html', user=user)
```

# 绑定多个数据库

从 0.12 开始，Flask-SQLAlchemy 可以容易地连接到多个数据库。为了实现这个功能，预配置了 SQLAlchemy 来支持多个 “binds”。

什么是绑定(binds)? 在 SQLAlchemy 中一个绑定(bind)是能执行 SQL 语句并且通常是一个连接或者引擎类的东东。在 Flask-SQLAlchemy 中，绑定(bind)总是背后自动为您创建好的引擎。这些引擎中的每个之后都会关联一个短键（bind key）。这个键会在模型声明时使用来把一个模型关联到一个特定引擎。

如果模型没有关联一个特定的引擎的话，就会使用默认的连接( `SQLALCHEMY_DATABASE_URI` 配置值)。

## 示例配置

下面的配置声明了三个数据库连接。特殊的默认值和另外两个分别名为 `users` （用于用户）和 `appmeta` 连接到一个提供只读访问应用内部数据的 sqlite 数据库）：

```
SQLALCHEMY_DATABASE_URI = 'postgres://localhost/main'
SQLALCHEMY_BINDS = {
    'users':        'mysqldb://localhost/users',
    'appmeta':      'sqlite:////path/to/appmeta.db'
}
```

## 创建和删除表

`create_all()` 和 `drop_all()` 方法默认作用于所有声明的绑定(bind)，包括默认的。这个行为可以通过提供 `bind` 参数来定制。它可以是单个绑定(bind)名, `'__all__'` 指向所有绑定(binds)或一个绑定(bind)名的列表。默认的绑定(bind)( `SQLALCHEMY_DATABASE_URI` ) 名为 `None` :

```
>>> db.create_all()
>>> db.create_all(bind=['users'])
>>> db.create_all(bind='appmeta')
>>> db.drop_all(bind=None)
```

## 引用绑定(Binds)

当您声明模型时，您可以用 `__bind_key__` 属性指定绑定(bind):

```
class User(db.Model):
    __bind_key__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
```

bind key 存储在表中的 `info` 字典中作为 `'bind_key'` 键值。了解这个很重要，因为当您想要直接创建一个表对象时，您会需要把它放在那:

```
user_favorites = db.Table('user_favorites',
    db.Column('user_id', db.Integer, db.ForeignKey('user.id')),
    db.Column('message_id', db.Integer, db.ForeignKey('message.id')),
    info={'bind_key': 'users'}
)
```

如果您在模型上指定了 `__bind_key__` ，您可以用它们准确地做您想要的。模型会自行连 接到指定的数据库连接。

# 信号支持

您可以订阅如下这些信号以便在更新提交到数据库之前以及之后得到通知。

如果配置中 `SQLALCHEMY_TRACK_MODIFICATIONS` 启用的话，这些更新变化才能被追踪。

New in version 0.10.

Changed in version 2.1: `before_models_committed` 正确地被触发。

Deprecated since version 2.1: 在以后的版本中，这个配置项默认是禁用的。

存在以下两个信号:

`models_committed`

这个信号在修改的模型提交到数据库时发出。发送者是发送修改的应用，模型和操作描述符以 `(model, operation)` 形式作为元组，这样的元组列表传递给接受者的 `changes` 参数。

该模型是发送到数据库的模型实例，当一个模型已经插入，操作是 `'insert'` ，而已删除是 `'delete'` ，如果更新了任何列，会是 `'update'` 。

`before_models_committed`

工作机制和 `models_committed` 完全一样，但是在提交之前发送。

# API

这部分文档记录了 Flask-SQLAlchemy 里的所有公开的类和函数。

## 配置

`class flask.ext.sqlalchemy.SQLAlchemy(app=None, use_native_unicode=True, session_options=N`

This class is used to control the SQLAlchemy integration to one or more Flask applications. Depending on how you initialize the object it is usable right away or will attach as needed to a Flask application.

There are two usage modes which work very similarly. One is binding the instance to a very specific Flask application:

```
app = Flask(__name__)
db = SQLAlchemy(app)
```

The second possibility is to create the object once and configure the application later to support it:

```
db = SQLAlchemy()

def create_app():
    app = Flask(__name__)
    db.init_app(app)
    return app
```

The difference between the two is that in the first case methods like `create_all()` and `drop_all()` will work all the time but in the second case a `flask.Flask.app_context()` has to exist.

By default Flask-SQLAlchemy will apply some backend-specific settings to improve your experience with them. As of SQLAlchemy 0.6 SQLAlchemy will probe the library for native unicode support. If it detects unicode it will let the library handle that, otherwise do that itself. Sometimes this detection can fail in which case you might want to set `use_native_unicode` (or the `SQLALCHEMY_NATIVE_UNICODE` configuration key) to `False`. Note that the configuration key overrides the value you pass to the constructor.

This class also provides access to all the SQLAlchemy functions and classes from the `sqlalchemy` and `sqlalchemy.orm` modules. So you can declare models like this:

```
class User(db.Model):
    username = db.Column(db.String(80), unique=True)
    pw_hash = db.Column(db.String(80))
```

You can still use `sqlalchemy` and `sqlalchemy.orm` directly, but note that Flask-SQLAlchemy customizations are available only through an instance of this `SQLAlchemy` class. Query classes default to `BaseQuery` for `db.Query`, `db.Model.query_class`, and the default query_class for `db.relationship` and `db.backref`. If you use these interfaces through `sqlalchemy` and `sqlalchemy.orm` directly, the default query class will be that of `sqlalchemy`.

Check types carefully

Don't perform type or `isinstance` checks against `db.Table`, which emulates `Table` behavior but is not a class. `db.Table` exposes the `Table` interface, but is a function which allows omission of metadata.

You may also define your own SessionExtension instances as well when defining your SQLAlchemy class instance. You may pass your custom instances to the `session_extensions` keyword. This can be either a single SessionExtension instance, or a list of SessionExtension instances. In the following use case we use the VersionedListener from the SQLAlchemy versioning examples.:

```
from history_meta import VersionedMeta, VersionedListener

app = Flask(__name__)
db = SQLAlchemy(app, session_extensions=[VersionedListener()])

class User(db.Model):
    __metaclass__ = VersionedMeta
    username = db.Column(db.String(80), unique=True)
    pw_hash = db.Column(db.String(80))
```

The `session_options` parameter can be used to override session options. If provided it's a dict of parameters passed to the session's constructor.

New in version 0.10: The `session_options` parameter was added.

New in version 0.16: `scopefunc` is now accepted on `session_options`. It allows specifying a custom function which will define the SQLAlchemy session's scoping.

New in version 2.1: The `metadata` parameter was added. This allows for setting custom naming conventions among other, non-trivial things.

`Query`

The `BaseQuery` class.

`apply_driver_hacks(app, info, options)`

This method is called before engine creation and used to inject driver specific hacks into the options. The `options` parameter is a dictionary of keyword arguments that will then be used to call the `sqlalchemy.create_engine()` function.

The default implementation provides some saner defaults for things like pool sizes for MySQL and sqlite. Also it injects the setting of `SQLALCHEMY_NATIVE_UNICODE`.

`create_all(bind='__all__', app=None)`

Creates all tables.

Changed in version 0.12: Parameters were added

`create_scoped_session(options=None)`

Helper factory method that creates a scoped session. It internally calls `create_session()`.

`create_session(options)`

Creates the session. The default implementation returns a `SignallingSession`.

New in version 2.0.

`drop_all(bind='__all__', app=None)`

Drops all tables.

Changed in version 0.12: Parameters were added

`engine`

Gives access to the engine. If the database configuration is bound to a specific application (initialized with an application) this will always return a database connection. If however the current application is used this might raise a `RuntimeError` if no application is active at the moment.

`get_app(reference_app=None)`

Helper method that implements the logic to look up an application.

`get_binds(app=None)`

Returns a dictionary with a table->engine mapping.

This is suitable for use of sessionmaker(binds=db.get_binds(app)).

`get_engine(app, bind=None)`

Returns a specific engine.

New in version 0.12.

`get_tables_for_bind(bind=None)`

Returns a list of all tables relevant for a bind.

`init_app(app)`

This callback can be used to initialize an application for the use with this database setup. Never use a database in the context of an application not initialized that way or connections will leak.

`make_connector(app, bind=None)`

Creates the connector for a given state and bind.

`make_declarative_base(model, metadata=None)`

Creates the declarative base.

`metadata`

Returns the metadata

`reflect(bind='__all__', app=None)`

Reflects tables from the database.

Changed in version 0.12: Parameters were added

# 模型

`class flask.ext.sqlalchemy.Model`

Baseclass for custom user models.

`__bind_key__`

Optionally declares the bind to use. `None` refers to the default bind. For more information see 绑定多个数据库.

`__tablename__`

The name of the table in the database. This is required by SQLAlchemy; however, Flask-SQLAlchemy will set it automatically if a model has a primary key defined. If the `__table__` or `__tablename__` is set explicitly, that will be used instead.

`query = None`

an instance of `query_class`. Can be used to query the database for instances of this model.

`query_class`

the query class used. The `query` attribute is an instance of this class. By default a `BaseQuery` is used.

alias of `BaseQuery`

`class flask.ext.sqlalchemy.BaseQuery(entities, session=None)`

The default query object used for models, and exposed as `Query`. This can be subclassed and replaced for individual models by setting the `query_class` attribute. This is a subclass of a standard SQLAlchemy `Query` class and has all the methods of a standard query as well.

`all()`

Return the results represented by this query as a list. This results in an execution of the underlying query.

`order_by(*criterion)`

apply one or more ORDER BY criterion to the query and return the newly resulting query.

`limit(limit)`

Apply a LIMIT to the query and return the newly resulting query.

`offset(offset)`

Apply an OFFSET to the query and return the newly resulting query.

`first()`

Return the first result of this query or `None` if the result doesn’t contain any rows. This results in an execution of the underlying query.

`first_or_404()`

Like `first()` but aborts with 404 if not found instead of returning `None`.

`get(ident)`

Return an instance based on the given primary key identifier, or `None` if not found.

E.g.:

```
my_user = session.query(User).get(5)

some_object = session.query(VersionedFoo).get((5, 10))
```

`get()` is special in that it provides direct access to the identity map of the owning `Session`. If the given primary key identifier is present in the local identity map, the object is returned directly from this collection and no SQL is emitted, unless the object has been marked fully expired. If not present, a SELECT is performed in order to locate the object.

`get()` also will perform a check if the object is present in the identity map and marked as expired - a SELECT is emitted to refresh the object as well as to ensure that the row is still present. If not, `ObjectDeletedError` is raised.

`get()` is only used to return a single mapped instance, not multiple instances or individual column constructs, and strictly on a single primary key value. The originating `Query` must be constructed in this way, i.e. against a single mapped entity, with no additional filtering criterion. Loading options via `options()` may be applied however, and will be used if the object is not yet locally present.

A lazy-loading, many-to-one attribute configured by `relationship()`, using a simple foreign-key-to-primary-key criterion, will also use an operation equivalent to `get()` in order to retrieve the target value from the local identity map before querying the database. See `/orm/loading_relationships` for further details on relationship loading.

Parameters: **ident** – A scalar or tuple value representing the primary key. For a composite primary key, the order of identifiers corresponds in most cases to that of the mapped `Table` object's primary key columns. For a `mapper()` that was given the `primary key` argument during construction, the order of identifiers corresponds to the elements present in this collection.

Returns: The object instance, or `None`.

`get_or_404(ident)`

Like `get()` but aborts with 404 if not found instead of returning `None`.

`paginate(page=None, per_page=None, error_out=True)`

Returns `per_page` items from page `page`. By default it will abort with 404 if no items were found and the page was larger than 1. This behavor can be disabled by setting `error_out` to `False`.

If page or per_page are None, they will be retrieved from the request query. If the values are not ints and `error_out` is true, it will abort with 404. If there is no request or they aren't in the query, they default to page 1 and 20 respectively.

Returns an `Pagination` object.

# 会话

`class flask.ext.sqlalchemy.SignallingSession(db, autocommit=False, autoflush=True, app=Non`

The signalling session is the default session that Flask-SQLAlchemy uses. It extends the default session system with bind selection and modification tracking.

If you want to use a different session you can override the `SQLAlchemy.create_session()` function.

New in version 2.0.

New in version 2.1: The `binds` option was added, which allows a session to be joined to an external transaction.

`app = None`

The application that this session belongs to.

# 实用工具

`class flask.ext.sqlalchemy.Pagination(query, page, per_page, total, items)`

Internal helper class returned by `BaseQuery.paginate()`. You can also construct it from any other SQLAlchemy query object if you are working with other libraries. Additionally it is possible to pass `None` as query object in which case the `prev()` and `next()` will no longer work.

`has_next`

True if a next page exists.

`has_prev`

True if a previous page exists

`items = None`

the items for the current page

`iter_pages(left_edge=2, left_current=2, right_current=5, right_edge=2)`

Iterates over the page numbers in the pagination. The four parameters control the thresholds how many numbers should be produced from the sides. Skipped page numbers are represented as `None`. This is how you could render such a pagination in the templates:

```
{% macro render_pagination(pagination, endpoint) %}
   class=pagination>
  {%- for page in pagination.iter_pages() %}
    {% if page %}
      {% if page != pagination.page %}
        <a href="{{ url_for(endpoint, page=page) }}">{{ page }}</a>
      {% else %}
        <strong>{{ page }}</strong>
      {% endif %}
    {% else %}
       class=ellipsis>…
    {% endif %}
  {%- endfor %}

{% endmacro %}
```

`next(error_out=False)`

Returns a `Pagination` object for the next page.

`next_num`

Number of the next page

`page = None`

the current page number (1 indexed)

`pages`

The total number of pages

`per_page = None`

the number of items to be displayed on a page.

`prev(error_out=False)`

Returns a `Pagination` object for the previous page.

`prev_num`

Number of the previous page.

`query = None`

the unlimited query object that was used to create this pagination object.

`total = None`

the total number of items matching the query

`flask.ext.sqlalchemy.get_debug_queries()`

In debug mode Flask-SQLAlchemy will log all the SQL queries sent to the database. This information is available until the end of request which makes it possible to easily ensure that the SQL generated is the one expected on errors or in unittesting. If you don't want to enable the DEBUG mode for your unittests you can also enable the query recording by setting the `'SQLALCHEMY_RECORD_QUERIES'` config variable to `True`. This is automatically enabled if Flask is in testing mode.

The value returned will be a list of named tuples with the following attributes:

`statement`

The SQL statement issued

`parameters`

The parameters for the SQL statement

`start_time` / `end_time`

Time the query started / the results arrived. Please keep in mind that the timer function used depends on your platform. These values are only useful for sorting or comparing. They do not necessarily represent an absolute timestamp.

`duration`

Time the query took in seconds

`context`

A string giving a rough estimation of where in your application query was issued. The exact format is undefined so don't try to reconstruct filename or function name.

# 更新历史

在这里您可以看到每一个 Flask-SQLAlchemy 发布版本的变化的完整列表。

## Version 2.1

In development

- Table names are automatically generated in more cases, including subclassing mixins and abstract models.

## Version 2.0

Released on August 29th 2014, codename Bohrium

- Changed how the builtin signals are subscribed to skip non Flask-SQLAlchemy sessions. This will also fix the attribute error about model changes not existing.
- Added a way to control how signals for model modifications are tracked.
- Made the `SignallingSession` a public interface and added a hook for customizing session creation.
- If the `bind` parameter is given to the signalling session it will no longer cause an error that a parameter is given twice.
- Added working table reflection support.
- Enabled autoflush by default.
- Consider `SQLALCHEMY_COMMIT_ON_TEARDOWN` harmful and remove from docs.

## Version 1.0

Released on July 20th 2013, codename Aurum

- Added Python 3.3 support.
- Dropped 2.5 compatibility.
- Various bugfixes
- Changed versioning format to do major releases for each update now.

## Version 0.16

- New distribution format (flask_sqlalchemy)
- Added support for Flask 0.9 specifics.

# Version 0.15

- Added session support for multiple databases

# Version 0.14

- Make relative sqlite paths relative to the application root.

# Version 0.13

- Fixed an issue with Flask-SQLAlchemy not selecting the correct binds.

# Version 0.12

- Added support for multiple databases.
- Expose Flask-SQLAlchemy's BaseQuery as `db.Query`.
- Set default query_class for `db.relation`, `db.relationship`, and `db.dynamic_loader` to Flask-SQLAlchemy's BaseQuery.
- Improved compatibility with Flask 0.7.

# Version 0.11

- Fixed a bug introduced in 0.10 with alternative table constructors.

# Version 0.10

- Added support for signals.
- Table names are now automatically set from the class name unless overriden.
- Model.query now always works for applications directly passed to the SQLAlchemy constructor. Furthermore the property now raises an RuntimeError instead of being None.
- added session options to constructor.
- fixed a broken `__repr__`
- `db.Table` is now a factor function that creates table objects. This makes it possible to omit the metadata.

# Version 0.9

- applied changes to pass the Flask extension approval process.

# Version 0.8

- added a few configuration keys for creating connections.
- automatically activate connection recycling for MySQL connections.
- added support for the Flask testing mode.

# Version 0.7

- Initial public release

# Flask-Testing

**Flask-Testing** 扩展为 Flask 提供了单元测试的工具。

## 安装 Flask-Testing

使用 **pip** 或者 **easy_install** 安装:

```
pip install Flask-Testing
```

或者从版本控制系统(github)中下载最新的版本:

```
git clone https://github.com/jarus/flask-testing.git
cd flask-testing
python setup.py develop
```

如果你正在使用 **virtualenv**，假设你会安装 **Flask-Testing** 在运行你的 Flask 应用程序的同一个 virtualenv 上。

## 编写测试用例

简单地继承 `TestCase` 的 MyTest:

```
from flask.ext.testing import TestCase

class MyTest(TestCase):

    pass
```

你必须定义 `create_app` 方法，该方法返回一个 Flask 实例:

```
from flask import Flask
from flask.ext.testing import TestCase

class MyTest(TestCase):

    def create_app(self):

        app = Flask(__name__)
        app.config['TESTING'] = True
        return app
```

如果不定义 `create_app` ，`NotImplementedError` 异常将会抛出。

## 使用 LiveServer 测试

如果你想要你的测试通过 Selenium 或者 无头浏览器(无头浏览器的意思就是无外设的意思，可以在命令行下运行的浏览器)运行，你可以使用 LiveServerTestCase:

```
import urllib2
from flask import Flask
from flask.ext.testing import LiveServerTestCase

class MyTest(LiveServerTestCase):

    def create_app(self):
        app = Flask(__name__)
        app.config['TESTING'] = True
        # Default port is 5000
        app.config['LIVESERVER_PORT'] = 8943
        return app

    def test_server_is_up_and_running(self):
        response = urllib2.urlopen(self.get_server_url())
        self.assertEqual(response.code, 200)
```

在这个例子中 `get_server_url` 方法将会返回 http://localhost:8943。

## 测试 JSON 响应

如果你正在测试一个返回 JSON 的视图函数的话，你可以使用 `Response` 对象的特殊的属性 `json` 来测试输出:

```
@app.route("/ajax/")
def some_json():
    return jsonify(success=True)

class TestViews(TestCase):
    def test_some_json(self):
        response = self.client.get("/ajax/")
        self.assertEquals(response.json, dict(success=True))
```

## 选择不渲染模板

当测试需要处理模板渲染的时候可能是一个大问题。如果在测试中你不想要渲染模板的话可以设置 `render_templates` 属性:

```
class TestNotRenderTemplates(TestCase):

    render_templates = False

    def test_assert_not_process_the_template(self):
        response = self.client.get("/template/")

        assert "" == response.data
```

尽管可以设置不想渲染模板，但是渲染模板的信号在任何时候都会发送，你也可以使用 `assert_template_used` 方法来检查模板是否被渲染:

```
class TestNotRenderTemplates(TestCase):

    render_templates = False

    def test_assert_mytemplate_used(self):
        response = self.client.get("/template/")

        self.assert_template_used('mytemplate.html')
```

当渲染模板被关闭的时候，测试执行起来会更加的快速并且视图函数的逻辑将会孤立地被测试。

## 使用 Twill

Twill 是一个用来通过使用命令行界面浏览网页的简单的语言。

Note

请注意 Twill 只支持 Python 2.x，不能在 Python 3 或者以上版本上使用。

`Flask-Testing` 拥有一个辅助类用来创建使用 Twill 的功能测试用例:

```
def test_something_with_twill(self):

    with Twill(self.app, port=3000) as t:
        t.browser.go(t.url("/"))
```

旧的 `TwillTestCase` 类已经被弃用。

## 测试 SQLAlchemy

这部分将会涉及使用 **Flask-Testing** 测试 SQLAlchemy 的一部分内容。这里假设你使用的是 Flask-SQLAlchemy 扩展，并且这里的例子也不是太难，可以适用于用户自己的配置。

首先，先确保数据库的 URI 是设置成开发环境而不是生产环境！其次，一个好的测试习惯就是在每一次测试执行的时候先创建表，在结束的时候删除表，这样保证干净的测试环境:

```
from flask.ext.testing import TestCase

from myapp import create_app, db

class MyTest(TestCase):

    SQLALCHEMY_DATABASE_URI = "sqlite://"
    TESTING = True

    def create_app(self):

        # pass in test configuration
        return create_app(self)

    def setUp(self):

        db.create_all()

    def tearDown(self):

        db.session.remove()
        db.drop_all()
```

同样需要注意地是每一个新的 SQLAlchemy 会话在测试用例运行的时候就被创建，`db.session.remove()` 在每一个测试用例的结尾被调用(这是为了确保 SQLAlchemy 会话及时被删除) - 这是一种常见的 “陷阱”。

另外一个 “陷阱” 就是 Flask-SQLAlchemy 会在每一个请求结束的时候删除 SQLAlchemy 会话(session)。因此每次调用 `client.get()` 或者其它客户端方法的后，SQLAlchemy 会话(session)连同添加到它的任何对象都会被删除。

例如:

```
class SomeTest(MyTest):

    def test_something(self):

        user = User()
        db.session.add(user)
        db.session.commit()

        # this works
        assert user in db.session

        response = self.client.get("/")

        # this raises an AssertionError
        assert user in db.session
```

你现在必须重新添加 “user” 实例回 SQLAlchemy 会话(session)使用 `db.session.add(user)`，如果你想要在数据库上做进一步的操作。

同样需要注意地是在这个例子中内存数据库 SQLite 是被使用：尽管它是十分的快，但是你要是使用其它类型的数据库(例如 MySQL 或者 PostgreSQL)，可能上述代码就不适用。

你也可能想要在 `setUp()` 里为你的数据库增加一组实例一旦你的数据库的表已经创建。如果你想要使用数据集的话，请参看 Fixture，它包含了对 SQLAlchemy 的支持。

# 运行测试用例

## 使用 unittest

一开始我建议把所有的测试放在一个文件里面，这样你可以使用 `unittest.main()` 函数。这个函数将会发现在你的 `TestCase` 类里面的所有的测试方法。请记住，所有的测试方法和类请以 `test` 开头（不区分大小写），这样才能被自动识别出来。

一个测试用例的文件可以看起来像这样:

```
import unittest
import flask.ext.testing

# your test cases

if __name__ == '__main__':
    unittest.main()
```

现在你可以用 `python tests.py` 命令执行你的测试。

## 使用 nose

同样 nose 也与 Flask-Testing 能够很好的融合在一起。

# 更新历史

## 0.4.2 (24.07.2014)

> >

> - Improved teardown to be more graceful.
> - Add `message` argument to `assertStatus` respectively all assertion methods with fixed status like `assert404`.

## 0.4.1 (27.02.2014)

This release is dedicated to every contributer who made this release possible. Thank you very much.

> >

> - Python 3 compatibility (without twill)
> - Add `LiveServerTestCase`
> - Use unittest2 backports if available in python 2.6
> - Install multiprocessing for python versions earlier than 2.6

## 0.4 (06.07.2012)

> >

> - Use of the new introduced import way for flask extensions. Use `import flask.ext.testing` instead of `import flaskext.testing`.
> - Replace all `assert` with `self.assert*` methods for better output with unittest.
> - Improved Python 2.5 support.
> - Use Flask's preferred JSON module.

# API

`class flask.ext.testing.TestCase(methodName='runTest')`

`assert200(response)`

Checks if response status code is 200

Parameters: **response** – Flask response

`assert400(response)`

Checks if response status code is 400

Versionadded: 0.2.5

Parameters: **response** – Flask response

`assert401(response)`

Checks if response status code is 401

Versionadded: 0.2.1

Parameters: **response** – Flask response

`assert403(response)`

Checks if response status code is 403

Versionadded: 0.2

Parameters: **response** – Flask response

`assert404(response)`

Checks if response status code is 404

Parameters: **response** – Flask response

`assert405(response)`

Checks if response status code is 405

Versionadded: 0.2

Parameters: **response** – Flask response

`assert500(response)`

Checks if response status code is 500

Versionadded: 0.4.1

Parameters: **response** – Flask response

`assertContext(name, value)`

Checks if given name exists in the template context and equals the given value.

Versionadded:

0.2

Parameters:

- **name** – name of context variable
- **value** – value to check against

`assertRedirects(response, location)`

Checks if response is an HTTP redirect to the given location.

Parameters:

- **response** – Flask response
- **location** – relative URL (i.e. without http://localhost)

`assertStatus(response, status_code)`

Helper method to check matching response status.

Parameters:

- **response** – Flask response
- **status_code** – response status code (e.g. 200)

`assertTemplateUsed(name, tmpl_name_attribute='name')`

Checks if a given template is used in the request. Only works if your version of Flask has signals support (0.6+) and blinker is installed. If the template engine used is not Jinja2, provide `tmpl_name_attribute` with a value of its `Template` class attribute name which contains the provided `name` value.

Versionadded:

0.2

Parameters:

- **name** – template name
- **tmpl_name_attribute** – template engine specific attribute name

`assert_200(response)`

Checks if response status code is 200

Parameters: **response** – Flask response

`assert_400(response)`

Checks if response status code is 400

Versionadded: 0.2.5

Parameters: **response** – Flask response

`assert_401(response)`

Checks if response status code is 401

Versionadded: 0.2.1

Parameters: **response** – Flask response

`assert_403(response)`

Checks if response status code is 403

Versionadded: 0.2

Parameters: **response** – Flask response

`assert_404(response)`

Checks if response status code is 404

Parameters: **response** – Flask response

`assert_405(response)`

Checks if response status code is 405

Versionadded: 0.2

Parameters: **response** – Flask response

`assert_500(response)`

Checks if response status code is 500

Versionadded: 0.4.1 Parameters: **response** – Flask response

`assert_context(name, value)`

Checks if given name exists in the template context and equals the given value.

Versionadded:

0.2

Parameters:

- **name** – name of context variable
- **value** – value to check against

`assert_redirects(response, location)`

Checks if response is an HTTP redirect to the given location.

Parameters:

- **response** – Flask response
- **location** – relative URL (i.e. without http://localhost)

`assert_status(response, status_code)`

Helper method to check matching response status.

Parameters:

- **response** – Flask response
- **status_code** – response status code (e.g. 200)

`assert_template_used(name, tmpl_name_attribute='name')`

Checks if a given template is used in the request. Only works if your version of Flask has signals support (0.6+) and blinker is installed. If the template engine used is not Jinja2, provide `tmpl_name_attribute` with a value of its `Template` class attribute name which contains the provided `name` value.

Versionadded:

0.2

Parameters:

- **name** – template name
- **tmpl_name_attribute** – template engine specific attribute name

`create_app()`

Create your Flask app here, with any configuration you need.

`get_context_variable(name)`

Returns a variable from the context passed to the template. Only works if your version of Flask has signals support (0.6+) and blinker is installed.

Raises a ContextVariableDoesNotExist exception if does not exist in context.

Versionadded: 0.2

Parameters: **name** – name of variable

`class flask.ext.testing.Twill(app, host='127.0.0.1', port=5000, scheme='http')`

Versionadded: 0.3

Twill wrapper utility class.

Creates a Twill `browser` instance and handles WSGI intercept.

Usage:

```
t = Twill(self.app)
with t:
    t.browser.go("/")
    t.url("/")
```

`url(url)`

Makes complete URL based on host, port and scheme Twill settings.

Parameters: **url** – relative URL

`class flask.ext.testing.TwillTestCase(methodName='runTest')`

Deprecated: use Twill helper class instead.

Creates a Twill `browser` instance and handles WSGI intercept.

`make_twill_url` (*url*)

Makes complete URL based on host, port and scheme Twill settings.

Parameters: **url** – relative URL

# Flask-WTF

**Flask-WTF** 提供了简单地 WTForms 的集成。

## 功能

- 集成 wtforms。
- 带有 csrf 令牌的安全表单。
- 全局的 csrf 保护。
- 支持验证码（Recaptcha）。
- 与 Flask-Uploads 一起支持文件上传。
- 国际化集成。

## 安装

该部分文档涵盖了 Flask-WTF 安装。使用任何软件包的第一步即是正确安装它。

### Distribute & Pip

用 pip 安装 Flask-WTF 是十分简单的:

```
$ pip install Flask-WTF
```

或者，使用 easy_install:

```
$ easy_install Flask-WTF
```

但是，你真的 不应该这样做。

### 获取源代码

Flask-WTF 在 GitHub 上活跃开发，代码在 GitHub 上 始终可用。

你也可以克隆公开仓库:

```
git clone git://github.com/lepture/flask-wtf.git
```

下载 tarball:

```
$ curl -OL https://github.com/lepture/flask-wtf/tarball/master
```

或者，下载 zipball:

```
$ curl -OL https://github.com/lepture/flask-wtf/zipball/master
```

当你有一份源码的副本后，你很容易地就可以把它嵌入到你的 Python 包中，或是安装到 site-packages:

```
$ python setup.py install
```

# 快读入门

急于上手？本页对 Flask-WTF 给出了一个详尽的介绍。假设你已经安装了 Flask-WTF，如果还未安装的话，请先浏览 安装。

## 创建表单

Flask-WTF 提供了对 WTForms 的集成。例如:

```
from flask_wtf import Form
from wtforms import StringField
from wtforms.validators import DataRequired

class MyForm(Form):
    name = StringField('name', validators=[DataRequired()])
```

Note

从 0.9.0 版本开始，Flask-WTF 将不会从 wtforms 中导入任何的内容，用户必须自己从 wtforms 中导入字段。

另外，隐藏的 CSRF 令牌会被自动地创建。你可以在模板这样地渲染它:

```
<form method="POST" action="/">
    {{ form.csrf_token }}
    {{ form.name.label }} {{ form.name(size=20) }}
    <input type="submit" value="Go">
</form>
```

尽管如此，为了创建有效的 XHTML/HTML， Form 类有一个 hidden_tag 方法， 它在一个隐藏的 DIV 标签中渲染任何隐藏的字段，包括 CSRF 字段:

```
<form method="POST" action="/">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name(size=20) }}
    <input type="submit" value="Go">
</form>
```

## 验证表单

在你的视图处理程序中验证请求:

```
@app.route('/submit', methods=('GET', 'POST'))
def submit():
    form = MyForm()
    if form.validate_on_submit():
        return redirect('/success')
    return render_template('submit.html', form=form)
```

注意你不需要把 `request.form` 传给 Flask-WTF；Flask-WTF 会自动加载。便捷的方法 `validate_on_submit` 将会检查是否是一个 POST 请求并且请求是否有效。

阅读 创建表单 学习更多的关于表单的技巧。

# 创建表单

这部分文档涉及表单(Form)信息。

## 安全表单

无需任何配置， `Form` 是一个带有 CSRF 保护的并且会话安全的表单。我们鼓励你什么都不做。

但是如果你想要禁用 CSRF 保护，你可以这样:

```
form = Form(csrf_enabled=False)
```

如果你想要全局禁用 CSRF 保护，你真的不应该这样做。但是你要坚持这样做的话，你可以在配置中这样写:

```
WTF_CSRF_ENABLED = False
```

为了生成 CSRF 令牌，你必须有一个密钥，这通常与你的 Flask 应用密钥一致。如果你想使用不同的密钥，可在配置中指定:

```
WTF_CSRF_SECRET_KEY = 'a random string'
```

# 文件上传

Flask-WTF 提供 `FileField` 来处理文件上传，它在表单提交后，自动从 `flask.request.files` 中抽取数据。 `FileField` 的 `data` 属性是一个 Werkzeug FileStorage 实例。

例如:

```
from werkzeug import secure_filename
from flask_wtf.file import FileField

class PhotoForm(Form):
    photo = FileField('Your photo')

@app.route('/upload/', methods=('GET', 'POST'))
def upload():
    form = PhotoForm()
    if form.validate_on_submit():
        filename = secure_filename(form.photo.data.filename)
        form.photo.data.save('uploads/' + filename)
    else:
        filename = None
    return render_template('upload.html', form=form, filename=filename)
```

### Note

记得把你的 HTML 表单的 `enctype` 设置成 `multipart/form-data` ，既是:

```
<form action="/upload/" method="POST" enctype="multipart/form-data">
    ....
</form>
```

此外，Flask-WTF 支持文件上传的验证。提供了 `FileRequired` 和 `FileAllowed` 。

`FileAllowed` 能够很好地和 Flask-Uploads 一起协同工作, 例如:

```
from flask.ext.uploads import UploadSet, IMAGES
from flask_wtf import Form
from flask_wtf.file import FileField, FileAllowed, FileRequired

images = UploadSet('images', IMAGES)

class UploadForm(Form):
    upload = FileField('image', validators=[
        FileRequired(),
        FileAllowed(images, 'Images only!')
    ])
```

也能在没有 Flask-Uploads 下挑大梁。这时候你需要向 `FileAllowed` 传入扩展名即可:

```
class UploadForm(Form):
    upload = FileField('image', validators=[
        FileRequired(),
        FileAllowed(['jpg', 'png'], 'Images only!')
    ])
```

# HTML5 控件

Note

自 wtforms 1.0.5 版本开始，wtforms 就内嵌了 HTML5 控件和字段。如果可能的话，你可以考虑从 wtforms 中导入它们。

我们将会在 0.9.3 版本后移除 html5 模块。

你可以从 `wtforms` 中导入一些 HTML5 控件:

```
from wtforms.fields.html5 import URLField
from wtforms.validators import url

class LinkForm(Form):
    url = URLField(validators=[url()])
```

## 验证码

Flask-WTF 通过 `RecaptchaField` 也提供对验证码的支持:

```
from flask_wtf import Form, RecaptchaField
from wtforms import TextField

class SignupForm(Form):
    username = TextField('Username')
    recaptcha = RecaptchaField()
```

这伴随着诸多配置，你需要逐一地配置它们。

| | |
|---|---|
| RECAPTCHA_PUBLIC_KEY | 必须 公钥 |
| RECAPTCHA_PRIVATE_KEY | 必须 私钥 |
| RECAPTCHA_API_SERVER | 可选 验证码 API 服务器 |
| RECAPTCHA_PARAMETERS | 可选 一个 JavaScript（api.js）参数的字典 |
| RECAPTCHA_DATA_ATTRS | 可选 一个数据属性项列表 https://developers.google.com/recaptcha/docs/display |

RECAPTCHA_PARAMETERS 和 RECAPTCHA_DATA_ATTRS 的示例:

```
RECAPTCHA_PARAMETERS = {'hl': 'zh', 'render': 'explicit'}
RECAPTCHA_DATA_ATTRS = {'theme': 'dark'}
```

对于应用测试时，如果 `app.testing` 为 `True` ，考虑到方便测试，Recaptcha 字段总是有效的。

在模板中很容易添加 Recaptcha 字段:

```
<form action="/" method="post">
    {{ form.username }}
    {{ form.recaptcha }}
</form>
```

我们为你提供了例子: recaptcha@github。

# CSRF 保护

这部分文档介绍了 CSRF 保护。

## 为什么需要 CSRF？

Flask-WTF 表单保护你免受 CSRF 威胁，你不需要有任何担心。尽管如此，如果你有不包含表单的视图，那么它们仍需要保护。

例如，由 AJAX 发送的 POST 请求，然而它背后并没有表单。在 Flask-WTF 0.9.0 以前的版本你无法获得 CSRF 令牌。这是为什么我们要实现 CSRF。

### 实现

为了能够让所有的视图函数受到 CSRF 保护，你需要开启 `CsrfProtect` 模块:

```
from flask_wtf.csrf import CsrfProtect

CsrfProtect(app)
```

像任何其它的 Flask 扩展一样，你可以惰性加载它:

```
from flask_wtf.csrf import CsrfProtect

csrf = CsrfProtect()

def create_app():
    app = Flask(__name__)
    csrf.init_app(app)
```

Note

你需要为 CSRF 保护设置一个秘钥。通常下，同 Flask 应用的 SECRET_KEY 是一样的。

如果模板中存在表单，你不需要做任何事情。与之前一样:

```
<form method="post" action="/">
    {{ form.csrf_token }}
</form>
```

但是如果模板中没有表单，你仍然需要一个 CSRF 令牌:

```
<form method="post" action="/">
    <input type="hidden" name="csrf_token" value="{{ csrf_token() }}" />
</form>
```

无论何时未通过 CSRF 验证，都会返回 400 响应。你可以自定义这个错误响应:

```
@csrf.error_handler
def csrf_error(reason):
    return render_template('csrf_error.html', reason=reason), 400
```

我们强烈建议你对所有视图启用 CSRF 保护。但也提供了某些视图函数不需要保护的装饰器:

```
@csrf.exempt
@app.route('/foo', methods=('GET', 'POST'))
def my_handler():
    # ...
    return 'ok'
```

默认情况下你也可以在所有的视图中禁用 CSRF 保护，通过设置 `WTF_CSRF_CHECK_DEFAULT` 为 `False`，仅仅当你需要的时候选择调用 `csrf.protect()`。这也能够让你在检查 CSRF 令牌前做一些预先处理:

```
@app.before_request
def check_csrf():
    if not is_oauth(request):
        csrf.protect()
```

# AJAX

不需要表单，通过 AJAX 发送 POST 请求成为可能。0.9.0 版本后这个功能变成可用的。

假设你已经使用了 `CsrfProtect(app)`，你可以通过 `{{ csrf_token() }}` 获取 CSRF 令牌。这个方法在每个模板中都可以使用，你并不需要担心在没有表单时如何渲染 CSRF 令牌字段。

我们推荐的方式是在 `<meta>` 标签中渲染 CSRF 令牌:

```
<meta name="csrf-token" content="{{ csrf_token() }}">
```

在 `<script>` 标签中渲染同样可行:

```
<script type="text/javascript">
    var csrftoken = "{{ csrf_token() }}"
</script>
```

下面的例子采用了在 `<meta>` 标签渲染的方式， 在 `<script>` 中渲染会更简单，你无须担心没有相应的例子。

无论何时你发送 AJAX POST 请求，为其添加 `X-CSRFToken` 头:

```
var csrftoken = $('meta[name=csrf-token]').attr('content')

$.ajaxSetup({
    beforeSend: function(xhr, settings) {
        if (!/^(GET|HEAD|OPTIONS|TRACE)$/i.test(settings.type) && !this.crossDomain) {
            xhr.setRequestHeader("X-CSRFToken", csrftoken)
        }
    }
})
```

# 故障排除

当你定义你的表单的时候，如果犯了 这个错误 _ : 从 `wtforms` 中导入 `Form` 而不是从 `flask.ext.wtf` 中导入，CSRF 保护的大部分功能都能工作(除了 `form.validate_on_submit()` )，但是 CSRF 保护将会发生异常。在提交表单的时候，你将会得到 `Bad Request` / `CSRF token missing or incorrect` 错误。这个错误的出现就是因为你的导入错误，而不是你的配置问题。

# 配置

这里是所有配置的全表。

# 表单和 CSRF

Flask-WTF 完整的配置清单。通常，你不必配置它们。默认的配置就能正常工作。

| | |
|---|---|
| WTF_CSRF_ENABLED | 禁用/开启表单的 CSRF 保护。默认是开启。 |
| WTF_CSRF_CHECK_DEFAULT | 默认下启用 CSRF 检查针对所有的视图。 默认值是 True。 |
| WTF_I18N_ENABLED | 禁用/开启 I18N 支持。需要和 Flask-Babel 配合一起使用。默认是开启。 |
| WTF_CSRF_HEADERS | 需要检验的 CSRF 令牌 HTTP 头。默认是 **['X-CSRFToken', 'X-CSRF-Token']** |
| WTF_CSRF_SECRET_KEY | 一个随机字符串生成 CSRF 令牌。 默认同 SECRET_KEY 一样。 |
| WTF_CSRF_TIME_LIMIT | CSRF 令牌过期时间。默认是 **3600** 秒。 |
| WTF_CSRF_SSL_STRICT | 使用 SSL 时进行严格保护。这会检查 HTTP Referrer， 验证是否同源。默认为 True 。 |
| WTF_CSRF_METHODS | 使用 CSRF 保护的 HTTP 方法。默认是 **['POST', 'PUT', 'PATCH']** |
| WTF_HIDDEN_TAG | 隐藏的 HTML 标签包装的名称。 默认是 **div** |
| WTF_HIDDEN_TAG_ATTRS | 隐藏的 HTML 标签包装的标签属性。 默认是 **{'style': 'display:none;'}** |

## 验证码

你已经在 验证码 中了解了这些配置选项。该表为了方便速查。

| | |
|---|---|
| RECAPTCHA_USE_SSL | 允许/禁用 Recaptcha 使用 SSL。默认是 False。 |
| RECAPTCHA_PUBLIC_KEY | 必须 公钥。 |
| RECAPTCHA_PRIVATE_KEY | 必须 私钥。 |
| RECAPTCHA_OPTIONS | 可选 配置选项的字典。 https://www.google.com/recaptcha/admin/create |

# 开发者接口

该部分文档涵盖了 Flask-WTF 的全部接口。

## 表单和字段

```
class flask_wtf.Form(formdata=<class flask_wtf.form._Auto at 0x10627ed50>, obj=None, prefi
```

Flask-specific subclass of WTForms **SecureForm** class.

If formdata is not specified, this will use flask.request.form. Explicitly pass formdata = None to prevent this.

Parameters:

- **csrf_context** – a session or dict-like object to use when making CSRF tokens. Default: flask.session.
- **secret_key** –

  a secret key for building CSRF tokens. If this isn't specified, the form will take the first of these that is defined:

  - SECRET_KEY attribute on this class
  - WTF_CSRF_SECRET_KEY config of flask app
  - SECRET_KEY config of flask app
  - session secret key
- **csrf_enabled** – whether to use CSRF protection. If False, all csrf behavior is suppressed. Default: WTF_CSRF_ENABLED config value

`hidden_tag(*fields)`

Wraps hidden fields in a hidden DIV tag, in order to keep XHTML compliance.

New in version 0.3.

Parameters: **fields** – list of hidden field names. If not provided will render all hidden fields, including the CSRF field.

`is_submitted()`

Checks if form has been submitted. The default case is if the HTTP method is **PUT** or **POST**.

`validate_csrf_data(data)`

Check if the csrf data is valid.

Parameters: **data** – the csrf string to be validated.

`validate_on_submit()`

Checks if form has been submitted and if so runs validate. This is a shortcut, equivalent to `form.is_submitted() and form.validate()`

`class flask_wtf.RecaptchaField(label='', validators=None, **kwargs)`

`class flask_wtf.Recaptcha(message=None)`

Validates a ReCaptcha.

`class flask_wtf.RecaptchaWidget`

`class flask_wtf.file.FileField(label=None, validators=None, filters=(), description=u'', i`

Werkzeug-aware subclass of **wtforms.FileField**

Provides a `has_file()` method to check if its data is a FileStorage instance with an actual file.

`has_file()`

Return True iff self.data is a FileStorage with file data

`class flask_wtf.file.FileAllowed(upload_set, message=None)`

Validates that the uploaded file is allowed by the given Flask-Uploads UploadSet.

Parameters:

- **upload_set** – A list/tuple of extention names or an instance of `flask.ext.uploads.UploadSet`
- **message** – error message

You can also use the synonym **file_allowed**.

`class flask_wtf.file.FileRequired(message=None)`

Validates that field has a file.

Parameters: **message** – error message

You can also use the synonym **file_required**.

`class flask_wtf.html5.SearchInput(input_type=None)`

Renders an input with type “search”.

`class flask_wtf.html5.SearchField(label=None, validators=None, filters=(), description=u''`

Represents an `<input type="search">`.

`class flask_wtf.html5.URLInput(input_type=None)`

Renders an input with type “url”.

`class flask_wtf.html5.URLField(label=None, validators=None, filters=(), description=u'', i`

Represents an `<input type="url">`.

`class flask_wtf.html5.EmailInput(input_type=None)`

Renders an input with type “email”.

`class flask_wtf.html5.EmailField(label=None, validators=None, filters=(), description=u'',`

Represents an `<input type="email">`.

`class flask_wtf.html5.TelInput(input_type=None)`

Renders an input with type “tel”.

`class flask_wtf.html5.TelField(label=None, validators=None, filters=(), description=u'', i`

Represents an `<input type="tel">` .

`class flask_wtf.html5.NumberInput(step=None)`

Renders an input with type “number”.

`class flask_wtf.html5.IntegerField(label=None, validators=None, **kwargs)`

Represents an `<input type="number">` .

`class flask_wtf.html5.DecimalField(label=None, validators=None, places=<unset value>, roun`

Represents an `<input type="number">` .

`class flask_wtf.html5.RangeInput(step=None)`

Renders an input with type “range”.

`class flask_wtf.html5.IntegerRangeField(label=None, validators=None, **kwargs)`

Represents an `<input type="range">` .

`class flask_wtf.html5.DecimalRangeField(label=None, validators=None, places=<unset value>,`

Represents an `<input type="range">` .

# CSRF 保护

`class flask_wtf.csrf.CsrfProtect(app=None)`

Enable csrf protect for Flask.

Register it with:

```
app = Flask(__name__)
CsrfProtect(app)
```

And in the templates, add the token input:

```
<input type="hidden" name="csrf_token" value="{{ csrf_token() }}"/>
```

If you need to send the token via AJAX, and there is no form:

```
<meta name="csrf_token" content="{{ csrf_token() }}" />
```

You can grab the csrf token with JavaScript, and send the token together.

`error_handler(view)`

A decorator that set the error response handler.

It accepts one parameter `reason` :

```
@csrf.error_handler
def csrf_error(reason):
    return render_template('error.html', reason=reason)
```

By default, it will return a 400 response.

`exempt(view)`

A decorator that can exclude a view from csrf protection.

Remember to put the decorator above the `route` :

```
csrf = CsrfProtect(app)

@csrf.exempt
@app.route('/some-view', methods=['POST'])
def some_view():
    return
```

`flask_wtf.csrf.generate_csrf(secret_key=None, time_limit=None)`

Generate csrf token code.

Parameters:

- **secret_key** – A secret key for mixing in the token, default is Flask.secret_key.
- **time_limit** – Token valid in the time limit, default is 3600s.

`flask_wtf.csrf.validate_csrf(data, secret_key=None, time_limit=None)`

Check if the given data is a valid csrf token.

Parameters:

- **data** – The csrf token value to be checked.
- **secret_key** – A secret key for mixing in the token, default is Flask.secret_key.
- **time_limit** – Check if the csrf token is expired. default is True.

# 升级到新版本

Flask-WTF 像其它软件一样随时间推移而改动。大多数改动是良性的，就是当你升级到新版而无需做出任何改动的良性。

尽管如此，每隔一段时间，就会有需要你对代码做出改动或是允许你改善你自己的代码质量来从 Flask-WTF 新特性获益的改动。

本节文档列举所有 Flask-WTF 版本中的所有变更以及如何进行无痛苦的升级。

如果你想用 pip 命令升级 Flask-WTF，确保传递 -U 参数:

```
$ pip install -U Flask-WTF
```

## 版本 0.9.0

移除 wtforms 的导入是一个重大的改变，这可能会给你带来许多痛苦，但这些导入项难以维护。你需要从原始的 WTForms 中导入 `Fields` ，而不是从 Flask-WTF 中导入:

```
from wtforms import TextField
```

配置选项 `CSRF_ENABLED` 改为 `WTF_CSRF_ENABLED` 。如果你没有设置任何配置选项，那么你无须做任何变动。

这个版本有很多的特色功能，如果你不需要他们，他们不会对你的任何代码有影响。

# Flask-WTF 更新历史

Flask-WTF 的所有发布版本的变更列表。

## Version 0.12

Released 2015/07/09

- Abstract protect_csrf() into a separate method
- Update reCAPTCHA configuration
- Fix reCAPTCHA error handle

## Version 0.11

Released 2015/01/21

- Use the new reCAPTCHA API via #164.

## Version 0.10.3

Released 2014/11/16

- Add configuration: WTF_CSRF_HEADERS via #159.
- Support customize hidden tags via #150.
- And many more bug fixes

# Version 0.10.2

Released 2014/09/03

- Update translation for reCaptcha via #146.

# Version 0.10.1

Released 2014/08/26

- Update RECAPTCHA API SERVER URL via #145.
- Update requirement Werkzeug>=0.9.5
- Fix CsrfProtect exempt for blueprints via #143.

# Version 0.10.0

Released 2014/07/16

- Add configuration: WTF_CSRF_METHODS
- Support WTForms 2.0 now
- Fix csrf validation without time limit (time_limit=False)
- CSRF exempt supports blueprint #111.

# Version 0.9.5

Released 2014/03/21

- `csrf_token` for all template types #112.
- Make FileRequired a subclass of InputRequired #108.

# Version 0.9.4

Released 2013/12/20

- Bugfix for csrf module when form has a prefix

- Compatible support for wtforms2
- Remove file API for FileField

# Version 0.9.3

Released 2013/10/02

- Fix validation of recaptcha when app in testing mode #89.
- Bugfix for csrf module #91

# Version 0.9.2

Released 2013/9/11

- Upgrade wtforms to 1.0.5.
- No lazy string for i18n #77.
- No DateInput widget in html5 #81.
- PUT and PATCH for CSRF #86.

# Version 0.9.1

Released 2013/8/21

This is a patch version for backward compitable for Flask<0.10 #82.

# Version 0.9.0

Released 2013/8/15

- Add i18n support (issue #65)
- Use default html5 widgets and fields provided by wtforms
- Python 3.3+ support
- Redesign form, replace SessionSecureForm
- CSRF protection solution
- Drop wtforms imports
- Fix recaptcha i18n support
- Fix recaptcha validator for python 3
- More test cases, it's 90%+ coverage now
- Redesign documentation

## Version 0.8.4

Released 2013/3/28

- Recaptcha Validator now returns provided message (issue #66)
- Minor doc fixes
- Fixed issue with tests barking because of nose/multiprocessing issue.

## Version 0.8.3

Released 2013/3/13

- Update documentation to indicate pending deprecation of WTForms namespace facade
- PEP8 fixes (issue #64)
- Fix Recaptcha widget (issue #49)

## Version 0.8.2 and prior

Initial development by Dan Jacob and Ron Duplain. 0.8.2 and prior there was not a change log.

# 作者

Flask-WTF 是由 Dan Jacob 创建，现在是由 Hsiaoming Yang 维护。

## 贡献者

贡献过补丁和建议的人们:

- Dan Jacob
- Ron DuPlain
- Daniel Lepage
- Anthony Ford
- Hsiaoming Yang

更多的贡献者可以在 GitHub 上找到。

# BSD 许可证